PATENT ABSTRACTS OF JAPAN

AA

(11)Publication number : 10-164140
(43)Date of publication of application : 19.06.1998

(51)Int.Cl. H04L 12/56
G06F 13/00
H04L 12/42

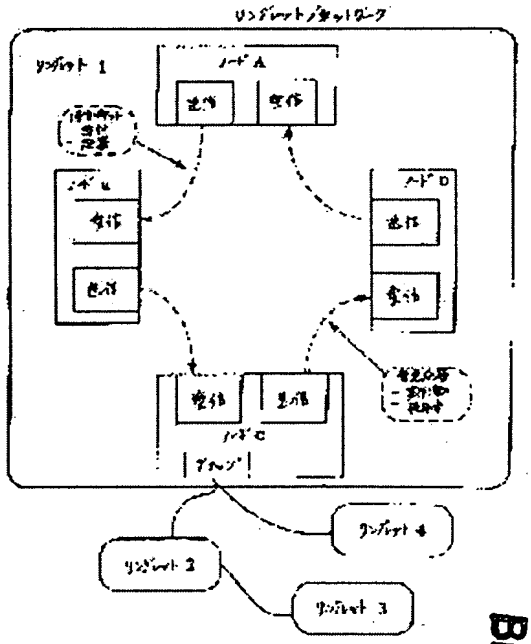
(21)Application number : 09-211428
(22)Date of filing : 01.07.1997

(71)Applicant : SUN MICROSYST INC
(72)Inventor : VAN LOO WILLIAM C

(30)Priority
Priority number : 96 673849 Priority date : 01.07.1996 Priority country : US

(54) SYSTEM FOR KEEPING FORCIBLY AND SEQUENTIALLY ORDERED PACKET FLOW IN RING NETWORK SYSTEM PROVIDED WITH BUSY NODE AND DEFECTIVE NODE

(57)Abstract:
PROBLEM TO BE SOLVED: To provide a system for keeping a reliable packet distribution in a ring network provided with the support of a forcibly ordered non-idenpotent command.
SOLUTION: Respective consumption side nodes in the network includes the final and previously known excellent packet and its order number record and holds the order of the packets passing through the nodes and the record of respective packet states at the time point of passage. When an error condition inside affirmation response concerning the packet is detected, a generation side node re-transmits the whole packets starting from the final and previously known excellent packet. The respective consumption side nodes processes or rejects the re-transmitted packets including the packet with the possibility of being the processed one. But it is recognized by the packet and state record concerning the whole packets.



LEGAL STATUS

[Date of request for examination] 08.06.2004
[Date of sending the examiner's decision of rejection]
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number]
[Date of registration]
[Number of appeal against examiner's decision of rejection]
[Date of requesting appeal against examiner's decision of rejection]

(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（Ａ）**　(11)特許出願公開番号

特開平10－164140

(43)公開日　平成10年(1998)６月19日

(51)Int.Cl.[6]	識別記号	ＦＩ	
Ｈ０４Ｌ　12/56		Ｈ０４Ｌ　11/20	１０２Ａ
Ｇ０６Ｆ　13/00	３５１	Ｇ０６Ｆ　13/00	３５１Ｍ
Ｈ０４Ｌ　12/42		Ｈ０４Ｌ　11/00	３３０

審査請求　未請求　請求項の数33　ＦＤ　外国語出願　（全202頁）

(21)出願番号　特願平9－211428

(22)出願日　平成９年(1997)７月１日

(31)優先権主張番号　０８／６７３８４９

(32)優先日　1996年７月１日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　591064003
サン・マイクロシステムズ・インコーポレーテッド
SUN MICROSYSTEMS, INCORPORATED
アメリカ合衆国　94303　カリフォルニア州・パロ　アルト・サン　アントニオ　ロード・901

(72)発明者　ウイリアム・シー・ヴァン・ルー
アメリカ合衆国・94301・カリフォルニア州・パロ　アルト・エマーソン　ストリート・2330

(74)代理人　弁理士　山川　政樹

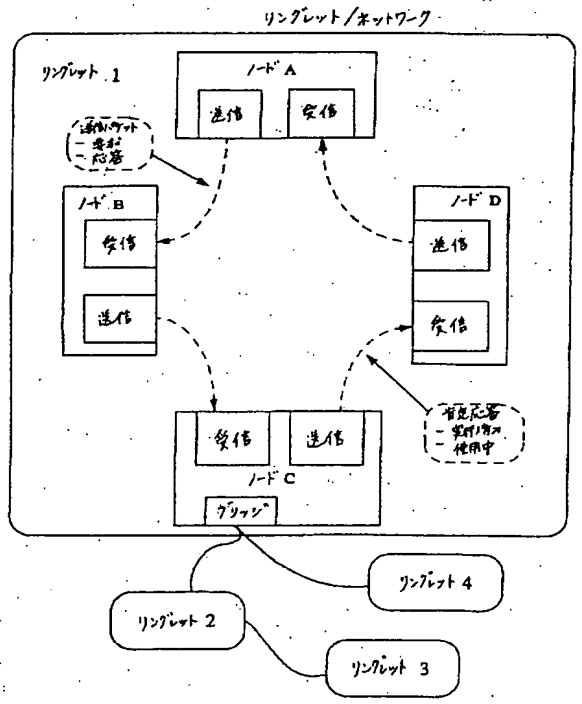
最終頁に続く

(54)【発明の名称】　ビジーノードと欠陥ノードを有するリング・ネットワーク・システムで強くシーケンシャルに順序付けられたパケット・フローを維持するシステム

(57)【要約】

【課題】　強力に順序付けられた非アイデンポテント・コマンドのサポートを備えたリング・ネットワーク内で信頼できるパケット配布を維持するためのシステムを提供する。

【解決手段】　ネットワーク上の各消費側ノードは、最後の既知の良好パケットとその順序番号の記録を含む、そのノードを通過したパケットの順序とそれを通過した時点でのそれぞれのパケットの状態の記録を保持している。作成側ノードは、パケットに関する肯定応答内のエラー条件を検出すると、最後の既知の良好パケットから始まるすべてのパケットを再送信する。各消費側ノードは、すでに処理された可能性のあるパケットを含む、再送信したパケットを処理または拒否することができるが、それについてはすべてのパケットに関するパケットおよび状態記録によって認識する。



1

【特許請求の範囲】

【請求項１】　コンピュータ・ネットワーク・サポート内の少なくとも１つの作成側ノードによって少なくとも１つの消費側ノードに伝送されるパケットの順序を維持するシステムにおいて、

前記作成側ノードによって送信された少なくとも１つの前記パケットに関する順序およびパケット状態情報を維持するように構成された前記作成側ノードの第１の送信サブシステムと、

前記消費側ノードによって前記作成側ノードに送信された肯定応答に関する順序およびパケット状態情報を維持し、送信したパケットが結果的に肯定応答になった条件を検出するように構成された前記作成側ノードの第１の受信サブシステムと、

前記消費側ノードからの使用中肯定応答を検出するように構成された前記作成側ノードの第２の受信サブシステムと、

前記肯定応答に関する順序およびパケット状態情報を維持するように構成された前記消費側ノードの第２の送信サブシステムと、

前記作成側ノードによって送信された前記パケットに関する順序およびパケット状態情報を維持し、前記作成側ノードから前記消費側ノードが受信したすべてのパケットに関する全体的な順序状態情報およびパケット受入れ状態情報を維持するように構成され、さらに少なくとも１つの前記使用中肯定応答が検出されたときにパケットを拒否するように構成された前記消費側ノードの第３の受信サブシステムと、

前記ネットワーク内の少なくとも１つの前記消費側ノードが所定の応答基準を満たすことができない場合を判定するように構成された前記作成側ノード内のノード監視サブシステムとを含むことを特徴とするシステム。

【請求項２】　前記ノード監視サブシステムが前記消費側ノードから受信した複数の使用中肯定応答を記憶するように構成された使用中肯定応答待ち行列を含んでいることを特徴とする請求項１に記載のシステム。

【請求項３】　前記ノード監視サブシステムが少なくとも所定数の前記使用中肯定応答を記憶している前記使用中肯定応答待ち行列に基づいて前記判定を行うように構成されていることを特徴とする請求項２に記載のシステム。

【請求項４】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから複数の使用中肯定応答を受信する所定長さの時間を含んでいることを特徴とする請求項１に記載のシステム。

【請求項５】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから使用中肯定応答を何も受信しない所定長さの時間を含んでいることを特徴とする請求項１に記載のシステム。

【請求項６】　前記システムが第１のパケットを２回以上受信したときに、前記第１のパケットがアドレス指定されているノードの状態を未変更の状態で維持しながら、非アイデンポテント要求を含む少なくとも前記第１のパケットを処理するように構成されていることを特徴とする請求項１に記載のシステム。

【請求項７】　前記再試行パケットの第２のインスタンスを受信したときに前記消費側ノードの状態を未変更の状態で維持しながら、非アイデンポテント要求を含む少なくとも１つのパケットを再試行する用に構成された再試行サブシステムを含むことを特徴とする請求項１に記載のシステム。

【請求項８】　前記ネットワークがリングレット・ネットワークであることを特徴とする請求項１に記載のシステム。

【請求項９】　前記ネットワークが強力順次順序付けを維持するように構成された少なくとも１つの順序付けされたノードと、強力順序付けを維持する態様以外の態様で構成された少なくとも１つの順序付けされていないノードとを含んでいることを特徴とする請求項１に記載のシステム。

【請求項１０】　前記ネットワークが非アイデンポテントコマンドをサポートするように構成された少なくとも第１のノードと、非アイデンポテントコマンドをサポートしないように構成された第２のノードとを含んでいることを特徴とする請求項１に記載のシステム。

【請求項１１】　前記ネットワークが異なる時点で、強力順次順序付けを維持する態様と、強力順次順序付けを維持する以外の態様の両方で構成できる少なくとも１つの動的ノードを含んでいることを特徴とする請求項１に記載のシステム。

【請求項１２】　エラー検出サブシステムと、

現行パケットの送信時に前記エラー検出サブシステムによってエラーを検出したときに、それぞれの前記消費側ノードの前記順序を共通ネットワーク値にリセットするように構成されたリセット・サブシステムとをさらに含むことを特徴とする請求項１に記載のシステム。

【請求項１３】　前記リセット・サブシステムが前記作成側ノードに以前送信したパケットを２回以上再送信させ、かつ有効な肯定応答パケットを前記作成側パケットが受信するまで、必要に応じ、これを繰り返させるように構成されていることを特徴とする請求項１２に記載のシステム。

【請求項１４】　前記の以前送信したパケットが肯定応答実行済みパケットを前記作成側ノードが受信したパケットであることを特徴とする請求項１３に記載のシステム。

【請求項１５】　前記の以前送信したパケットと前記現行パケットを含んで、これらの間で再試行パケットを送信するように構成されている第１の再試行サブシステムをさらに含んでいることを特徴とする請求項１４に記載

のシステム。

【請求項１６】　前記再試行パケットの送信に応答して前記作成側ノードで受信した再試行パケット肯定応答の妥当性を判定するように構成された再試行パケット妥当性検査サブシステムをさらに含んでいることを特徴とする請求項１５に記載のシステム。

【請求項１７】　再試行パケット肯定応答が無効状態であると前記再試行パケット妥当性検査サブシステムが判定した場合に、前記再試行パケットを送信するように構成された第２の再試行サブシステムをさらに含んでいることを特徴とする請求項１６に記載のシステム。

【請求項１８】　前記リセット・サブシステムが再送信前に前記の以前に送信したパケットからデータ・フィールドを除去するように構成されていることを特徴とする請求項１３に記載のシステム。

【請求項１９】　前記リセット・サブシステムが前記の各消費側ノードにおいて順序妥当状態値を維持し、前記の以前に送信したパケットの前記の各消費側ノードにおける受信時に前記の各順序妥当状態値を共通値にリセットするように構成された順序妥当状態サブシステムを含んでいることを特徴とする請求項１３に記載のシステム。

【請求項２０】　前記再試行サブシステムが前記の各消費側ノードにおいて受入れ妥当状態値を維持するように構成された受入れ妥当状態サブシステムと、
前記再試行パケットの受入れ有効フィールドと前記の各消費側ノードにおける前記受入れ妥当状態値との比較を生成するように構成された受入れ妥当性比較サブシステムと、
前記の比較が所定の基準に合致している前記の各再試行パケットを拒絶する再試行パケット拒絶サブシステムとを含んでいることを特徴とする請求項１５に記載のシステム。

【請求項２１】　複数の送信および消費側ノードを含み、それぞれの前記消費側ノードが、前記パケットおよび肯定応答が前記状態情報を読み取る消費側ノード以外のノードにアドレス指定されていても、強力順次順序付けをサポートするすべてのノード間で共通の順序付け情報を維持するために、複数の前記パケットおよび前記肯定応答の状態情報をそれぞれ読み取るように構成された前記受信サブシステムの１つを含むことを特徴とする請求項１に記載のシステム。

【請求項２２】　複数の前記作成側ノードと複数の前記消費側ノードとを含み、
前記作成側および消費側ノードの少なくとも部分集合が、ＳＳＯパケットをそれぞれ送信および受信し、前記ＳＳＯパケットのＳＳＯ肯定応答をそれぞれ受信および送信するための強力順次順序付け（ＳＳＯ）ノードとして構成され、
前記部分集合のそれぞれの前記作成側および消費側ノードが、前記ＳＳＯパケットまたは前記ＳＳＯ肯定応答あるいはその両方を読み取るように構成されていることを特徴とする請求項１に記載のシステム。

【請求項２３】　前記ネットワーク上を作成側ノードによって伝送された際の要求パケットの順序を未変更の状態に維持し、かつ前記要求パケットに応じて消費側ノードによって前記ネットワーク上に生成された応答パケットの順序を未変更の状態に維持するように構成されていることを特徴とする請求項１に記載のシステム。

【請求項２４】　コンピュータ・ネットワーク内の少なくとも１つの作成側ノードによって少なくとも１つの消費側ノードに伝送されるパケットの順序を維持するためのシステムにおいて、
その順序内の少なくとも１つのパケットがその順序内の有効な位置にあるかどうかを判定するように構成された前記消費側ノード内の順序検査サブシステムと、
受信した肯定応答が前記消費側ノード内の使用中条件を示すかどうかを判定するように構成された前記作成側ノード内の使用中肯定応答検出サブシステムと、
前記ネットワーク内の少なくとも１つの前記消費側ノードが所定の応答基準を満たすことができない場合を判定するように構成された前記作成側ノード内のノード監視サブシステムとを含むことを特徴とするシステム。

【請求項２５】　前記ノード監視サブシステムが前記消費側ノードから受信した複数の使用中肯定応答を記憶するように構成された使用中肯定応答待ち行列を含んでいることを特徴とする請求項２４に記載のシステム。

【請求項２６】　前記ノード監視サブシステムが少なくとも所定数の前記使用中肯定応答を記憶している前記使用中肯定応答待ち行列に基づいて前記判定を行うように構成されていることを特徴とする請求項２５に記載のシステム。

【請求項２７】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから複数の使用中肯定応答を受信する所定長さの時間を含んでいることを特徴とする請求項２４に記載のシステム。

【請求項２８】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから使用中肯定応答を何も受信しない所定長さの時間を含んでいることを特徴とする請求項２４に記載のシステム。

【請求項２９】　コンピュータ・ネットワーク内の少なくとも１つの作成側ノードによって少なくとも１つの消費側ノードに伝送されるパケットの順序を維持するためのシステムにおいて、
前に伝送され未処理のパケットに関する前記作成側ノードへの肯定応答がない場合に前記作成側ノードからのパケットをパイプライン化するように構成されたパイプライン化サブシステムと、
第１の前記未処理パケットと現行パケットとの間の順序カウント差を決定するように構成されたカウンタ・サブ

システムと、
前記順序カウント差が所定のしきい値に達したときに前記パケットの前記パイプライン化を終了するように構成された遮断サブシステムと、
前記消費側ノードで使用中条件を検出するように構成された使用中検出サブシステムと、
前記使用中条件が検出されたときに前記パケットのパイプライン化を中断するように構成されたパケット中断サブシステムと、
前記ネットワーク内の少なくとも１つの前記消費側ノードが所定の応答基準を満たすことができない場合を判定するように構成された前記作成側ノード内のノード監視サブシステムとを含むことを特徴とするシステム。
【請求項３０】　前記ノード監視サブシステムが前記消費側ノードから受信した複数の使用中肯定応答を記憶するように構成された使用中肯定応答待ち行列を含んでいることを特徴とする請求項２９に記載のシステム。
【請求項３１】　前記ノード監視サブシステムが少なくとも所定数の前記使用中肯定応答を記憶している前記使用中肯定応答待ち行列に基づいて前記判定を行うように構成されていることを特徴とする請求項３０に記載のシステム。
【請求項３２】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから複数の使用中肯定応答を受信する所定長さの時間を含んでいることを特徴とする請求項２９に記載のシステム。
【請求項３３】　前記所定の応答基準が前記作成側ノードが前記消費側ノードから使用中肯定応答を何も受信しない所定長さの時間を含んでいることを特徴とする請求項２９に記載のシステム。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】本発明は、直列相互接続を使用するプロセッサ・ベースのネットワーク内でパケット送信をサポートするためのシステムに関する。具体的には、本発明のシステムは、このようなネットワーク内、特にリングレット・トポロジ内のエラーに応答してパケットの動的順序付けをサポートし、過負荷またはノード障害のいずれかにより応答できないかまたは長時間の間、使用中肯定応答によって応答するノードと、ネットワーク上のノードにおける使用中条件の両方に対応する。
【０００２】
【従来の技術】コンピュータ・システム内の直列相互接続は、いくつかの様々なタイプのサービス割込みの影響を受ける可能性がある。たとえば、ネットワーク上のノードがパケット内のＣＲＣ（巡回冗長検査）エラーを検出した場合、そのパケットは受け入れることができない。そのパケットを送信したノードは、一般に間接的に（たとえば、タイムアウトによるか、アイドル・パケットの使用による）エラーを認識し、結局、そのパケットを再送信しなければならない。
【０００３】特に、ネットワークが緩和メモリ順序付け（ＲＭＯ）、強力順次順序付け（ＳＳＯ）、その他または中間厳密度の順序付けなどの順序付け方式を実施する場合には、パケットの再送信が単純なことではない可能性がある。このような順序付け方式を備えたパケット交換ネットワークでは、ＣＲＣエラーを発生するパケットの前後のパケットは、作成側ノードからターゲット・ノードに再送信する必要がある可能性がある。このようなパケットが非アイデンポテント・コマンド、すなわち、ターゲット・ノードで実行すると、そのノードでコマンドを再実行したときに第１の実行とは異なる結果が得られるようにそのノードの状態を変更するコマンドを含む場合、特定の問題が発生する。この場合、コマンドをそのノードに再送信し、もう一度実行した場合、不要または予測できない結果が発生する可能性がある。
【０００４】
【発明が解決しようとする課題】したがって、アイデンポテント・コマンドのサポートを維持しながら、ターゲット・ノードにパケットを再送信することによってパケットのエラーに対応可能なシステムが必要である。特に、様々なレベルの順序付け方式もサポートするようなシステムが必要である。
【０００５】挙動不良のノード、すなわち、故障しているかまたは応答するのに不当に長い時間を要するノードに対応できるようなシステムに対する特定の必要性が存在する。単一受信側の挙動が不適切なトポロジでは、このようなノードは、新しいパケットが導入される前に同じ使用中パケットの反復送信を強制的に完了させることにより、新しいパケットの順方向の進行を妨げる恐れがある。したがって、このような潜在的な落とし穴を処理するシステムが必要である。
【０００６】
【課題を解決するための手段】すべての既知の良好パケットの各受信ノードで状態を維持することにより、使用中ａｃｋを処理すると同時に結果的にＣＲＣエラーを発生しているパケットの再送信を行うシステムを提示する。パケットを再送信する必要がある場合、ローカル・ノードは、再送信されたパケットをすでに処理したかどうかを把握し、どのパケットが最後の既知の良好パケットであるかを把握し、このため、非アイデンポテント・コマンドを含む、すでに実行されたコマンドの再処理を回避することができる。使用中ループはエラー・ループが実行されている間に効率よく中断することができ、パケットの再順序付けが行われると、使用中ループを完了することができる。さらに、システムは、作成側ノードに応答を返送しないような挙動不良、たとえば、非応答ノードに対応する。このような挙動不良ノードは、待ち行列化した再試行方針によって効率よく中立化すること

ができる。この場合、何らかの所定のしきい値に達するまで再試行パケット待ち行列に使用中パケットが蓄積され、その後、作成側ノードがそのノードへのパケットを再試行しようというその試みを終了することにより、継続的に使用中のノードが効率よくシステムから除去される。このようにして、システムは他の未解決パケットの処理を続行することができる。

【０００７】したがって、本出願は、リングレット・ネットワーク上で同時に発生する３通りの問題、すなわち、エラー条件、１つまたは複数のノードでの使用中条件、ノードの故障または過負荷の解決を達成する複雑なシステムに関する。適当なエラー再試行メカニズムについては、本明細書と、ｖａｎ　Ｌｏｏ他により１９９６年７月１日に出願され、Ｓｙｓｔｅｍ　ｆｏｒ　Ｄｙｎａｍｉｃ　Ｏｒｄｅｒｉｎｇ　Ｓｕｐｐｏｒｔ　ｉｎ　ａ　Ｒｉｎｇｌｅｔ　Ｓｅｒｉａｌ　Ｉｎｔｅｒｃｏｎｎｅｃｔという名称の本出願人の関連特許出願の両方に記載されている。このようなエラー再試行メカニズムに基づくと同時に使用中再試行動作を処理できるシステムについては、ｖａｎ　Ｌｏｏ他により１９９６年７月１日に出願され、Ｓｙｓｔｅｍ　ｆｏｒ　Ｐｒｅｓｅｒｖｉｎｇ　Ｓｅｑｕｅｎｔｉａｌ　Ｏｒｄｅｒｉｎｇ　ａｎｄ　Ｓｕｐｐｏｒｔｉｎｇ　Ｉｄｅｍｐｏｔｅｎｔ　Ｃｏｍｍａｎｄｓ　ｉｎ　ａ　Ｒｉｎｇ　Ｎｅｔｗｏｒｋ　ｗｉｔｈ　Ｂｕｓｙ　Ｎｏｄｅｓという名称の本出願人の関連特許出願に記載されている。これらの特許出願はいずれも参照により本明細書に組み込まれる。

【０００８】

【発明の実施の形態】図１は、それぞれが送受信機能を備えた４つのノードＡ～Ｄを含む標準的なリングレット１を示している。要求および応答は作成側ノード、たとえば、ノードＡによって送出され、肯定応答（たとえば、ａｃｋ＿ｄｏｎｅおよびａｃｋ＿ｂｕｓｙ）は消費側ノード、たとえば、ノードＣによって送信される。ノードＣは従来通り他のリングレット２、３、４と通信することもできる。本発明は、特に、リングレット１などのリング・ネットワーク内のエラーの影響を受けるパケットの解決策に関する。これは予測不能なパケットの配達および受入れを発生する可能性がある。というのは、作成側はそれが何を送信したかを把握しているが、肯定応答が受信されていないパケットが実際に意図されたノードによって処理されたかどうかを把握していない（間接手段によるものを除く）からである。

【０００９】順序付けが必要な場合、課題はさらに大きくなる。図２は、ノードＡによって特定の順序Ａ、Ｂ、Ｃ・・・で送信されるコマンドと、リングレットおよび送信時の変動のために、Ｂ、Ａ、Ａ、Ａ・・・などの他の順序での受信とを示している。システムの挙動が適切であれば、送信パケットを再順序付けし、最初に送信された順序でその送信パケットを適切に実行できるはずである。

【００１０】図３は強力順次順序付け（ＳＳＯ）を示している。すなわち、いずれのパケットも送信されたときの順序から外れて受信されることはなく、２回現れる可能性はあるが、順不同で現れることはない。図４はＳＳＯと非アイデンポテント・コマンドの両方の使用を示している。順序が保持されるだけでなく、非アイデンポテントコマンドの特性により、従来システム内のコマンドはいずれも宛先ノードによって２回実行することはできない。ただし、図４において、｛ｐｒｏｄｕｃｅｒＩｄ，　ｐＬａｂｅｌ｝はｓｅｎｄ＿ｐｋｔおよび肯定応答に共通し、また、｛ｐｒｏｄｕｃｅｒＩｄ，　ｐＬａｂｅｌ｝は各ノードから見えるもので、不在エラーであることに留意されたい。

【００１１】図５は上記の課題に対する本出願人の解決策に適したデータ構造を示している。パケットはｐｒｏｄｕｃｅｒＩＤとｐＬａｂｅｌというフィールドを含むが、詳細については以下に述べる。この２つのフィールドは、消費側から作成側に返送される肯定応答パケット内に含まれる。

【００１２】本発明は、図６に示すようなパターンで既知の良好な（すでに受け入れられた／実行済みの）パケットとそれ以外のパケットの両方を再送信する。図６Ａは本発明のシステムの基本的な手法を高レベルで要約したものであり、有効肯定応答（「実行済み」の場合もあれば「使用中」の場合もある）が作成側ノードで受信されるまで、必要に応じて既知の良好ＳＳＯパケットを繰り返し再送信する。次に作成側ノードは、適切に処理されたもの、すなわち、それに関してａｃｋ＿ｄｏｎｅが受信されたものであっても、既知の良好パケットの後で、他のすべてのパケットを再送信する。（図７を参照。）ａｃｋは再送信パケットから戻ってくるので、作成側ノードは、すべてが有効ａｃｋ、すなわち、「実行済み」または「使用済み」であることを確認するために検査する。すべてが有効ａｃｋであるわけではない場合、すべてのａｃｋが実際に有効になるまで再送信プロセスを繰り返す。これにより、すべてのエラー・パケットが適切に処理されたことが確認され、システムは後続パケットについて処理を続行することができる。この手法では、ＳＳＯを保持し、非アイデンポテント・コマンドをサポートするために、複雑な課題が提示され、図９～３９に示す論理によって解決策が提示される。その動作については図４０以降の状態図に示す。

【００１３】本発明は、消費側ノードがパケット・アクションに関する状態（受入れおよび拒否状態）とａｃｋに関する状態（実行済みおよび使用中）を継続的に維持するという事実によって使用可能になる。これは、各ノードが検出するすべてのパケットおよびａｃｋに当てはまる。したがって、各消費側ノードは、各パケットがそのノードを通過したときにリングレットを移動するすべ

てのパケットの処分を認識している。この情報は再送信プロセスでは重要なものである。しかも、各消費側ノードは、通過したパケットの順序と、最後の既知の良好パケット（それに関してａｃｋ＿ｄｏｎｅまたはａｃｋ＿ｂｕｓｙが出されたもの）の順序番号の記録を保持している。これは、ＣＲＣエラーなどが検出されたときに必要であればシステムがバックアップ可能な点を示すものである。これについては、図４０以降の例に関する以下の説明でより明確に示すものとする。

【００１４】図８は、本発明に適したハードウェア環境を示すブロック図であり、作成側ノード内で可能なパケットの流れを示すためにも役に立つものである。これは、図４０以降の流れ図／状態図および図９～３９の論理図を考慮すると、詳細に理解することができる。

【００１５】図４０には、システムの基本ブロックが示されている。ＳＳＯ順序付け拡張機能がない場合、このシステムは、以下に示す削除および変更を伴うすべてのブロックを含むものと思われる。

－ｒｅｔｒｙ＿ｐｋｔレジスタ７－１９０を削除する

－ｒｅｔｒｙ＿ｐｋｔ待ち行列７－３０を削除する

－７－１００内のｏｒ＿ｌａｓｔ＿ｃｔカウンタを削除し、ｎｏ＿ｌａｓｔ＿ｃｔを保持する

－４：１マルチプレクサを３：１マルチプレクサ７－４０に変更する

－ｏｒ＿ｒｃｖｄ＿ｃｔカウンタ７－２５０を削除する

その他のブロックは、ＳＳＯサポートを伴わない送信ノードの動作にとって依然として基本的なものである。

【００１６】図８と図４０の詳細ブロック図の両方を参照し、システムの基本動作を以下に示す。要求および応答ＦＩＦＯ待ち行列によりトランザクション層からリンク層にパケットを挿入する。図示の通り、ｓｅｎｄ＿ｐｋｔレジスタ、ＣＲＣジェネレータと８／２０エンコーダ、シリアライザによるリングレットへの（可能な）送信のためにマルチプレクサ（ＭＵＸ）によりパケットを選択する。

【００１７】このノードからの要求および応答送信パケットに関する肯定応答ならびに他のノードから発信されたがこのノードに向けられている着信要求および応答パケットをデシリアライザで受信する。８／１０デコードおよびＣＲＣ検査の後、着信受信および肯定応答パケットをｒｅｃ＿ｐｋｔレジスタに登録する。ｒｅｃ＿ｐｋｔレジスタの前にアイドル記号を取り除く。ＣＲＣエラーまたは８／１０エンコード・エラーにより、着信受信および肯定応答パケットが無効になる。ＣＲＣエラーを伴う肯定応答はｒｅｃ＿ｐｋｔレジスタを越えて目に見えるものではないので、他のノードから受信した送信パケットは、トランザクション層にとって見えないように処理する。（受信バッファの説明を参照。）ａｃｋデコード論理は、それがそのシリアライザにより送信した送信パケットに関してこのノードにアドレス指定されたａｃｋを検出するものである。

【００１８】ａｃｋは、ａｃｋ待ち行列内に待ち行列化し、ａｃｋ＿ｌｏｇｉｃブロック内でｓｅｎｄ＿ｐｋｔ待ち行列内の送信待ち行列と突き合わせる。ＳＳＯサポートがない場合、すべてのパケットはｎｏｎ＿ｏｒｄｅｒレジスタを通って流れるはずであり、ＳＳＯの場合は、すべてのＳＳＯパケットがｒｅｔｒｙ＿ｐｋｔレジスタを通って流れ、非ＳＳＯパケットはｎｏｎ＿ｏｒｄｅｒレジスタにより継続する。

【００１９】再送信論理は、基本的にａｃｋ状況を検査し、パケットが完了したか（ａｃｋ＿ｄｏｎｅ）または受信側が使用中であるために再試行が必要であるか（ａｃｋ＿ｂｕｓｙ）を判定する。ａｃｋ＿ｄｏｎｅパケットは、「コマンド・レベルへのａｃｋ」ブロックによりトランザクション層に状況を返す。ＭＵＸ論理および４：１（または、本発明の指定の実施形態に応じて３：１）ＭＵＸにより再試行するために、非ＳＳＯ使用中再試行パケットを選択することができる。ＳＳＯ使用中再試行パケットはｒｅｔｒｙ＿ｐｋｔＦＩＦＯ待ち行列に待ち行列化される。このようなパケットは、ＳＳＯ使用中再試行論理によってＭＵＸ論理およびＭＵＸを介してこの待ち行列から選択される。これについては、システム全体に分散され、図１０～２５の送信論理用の流れ図に記載する。

【００２０】図１０～２６の送信論理は、基本的に以下のように動作する。その動作の詳細は、図４０以降の状態図／流れ図に関連して流れの例に関して以下により明確になるだろう。以下の説明の各論理ユニット（たとえば、ａｃｋ＿ｌｏｇｉｃ）は図４０内のこのような要素を指し示す。

【００２１】図１０は、エラー条件の場合に肯定応答と照らし合わせてパケットをテストし、エラーがない場合に使用中再試行を開始するためのａｃｋ論理である。

【００２２】図１１は、エラーがない場合に通常パケット処理のためにａｃｋ論理出力を設定するためのａｃｋ論理である。

【００２３】図１２は、ｎｏｎ＿ｏｒｄｅｒレジスタにロードすべき非ＳＳＯパケット用のａｃｋ論理内の処理である。

【００２４】図１３は、ＣＲＣ＿ｉｎｉｔ＿ｅｒｒエラー再試行ループを処理するａｃｋ論理であり、それがこのパケット用の有効（実行済みまたは使用中）肯定応答を受信するまで最後の「既知の良好」ＳＳＯパケットを再送信するものである。

【００２５】図１４は、ＣＲＣ＿ｅｒｒエラー再試行ループを処理するａｃｋ論理であり、リングレットにより「既知の良好」パケット以降のすべてのパケットを再送信するものである。

【００２６】図１５は、ＣＲＣ＿ａｃｋ＿ｃｈｋエラー再試行ループを処理するａｃｋ論理であり、ＣＲＣ＿ｅ

ｒｒループで再試行されたすべてのパケットがエラーなしで完了したかまたはエラーが発生しているかをテストするものである。エラーが検出された場合、ＣＲＣ＿ｉｎｉｔ＿ｅｒｒからループを再試行する（図３１）。エラーが一切検出されない場合、ＣＲＣ＿ｅｒｒ＿ｅｎｄを設定する。

【００２７】図１６は、ＣＲＣ＿ａｃｋ＿ｃｈｋエラー再試行ループを処理するａｃｋ論理であり、ＣＲＣ＿ｅｒｒループ中に送信した各パケットの肯定応答について有効肯定応答（ａｃｋ＿ｂｕｓｙまたはａｃｋ＿ｄｏｎｅ）があるかどうかをテストし（図３２）、エラーが検出された場合にエラー・ビットをを設定する。

【００２８】図１７は、パケット状態情報のフォーマットを処理し、ＣＲＣ＿ｉｎｉｔ＿ｅｒｒ、ＣＲＣ＿ｅｒｒ、ＣＲＣ＿ａｃｋ＿ｃｈｋループの実行中にＭＵＸを設定する再送信論理である。

【００２９】図１８は、ＣＲＣ＿ｅｒｒ＿ｅｎｄ制御を完了するための条件を検出し、ｒｅｔｒｙ＿ｐｋｔ待ち行列が空であるかどうかをテストすることにより、使用中再試行ループの完了を検出するためにｂｕｓｙ＿ｌｏｏｐ＿ｃｍｐｌｔ制御を設定するための再送信論理である。

【００３０】図１９は、実行済み（ａｃｋ＿ｄｏｎｅ）または使用中再試行を必要とするもの（ａｃｋ＿ｂｕｓｙ）としてＳＳＯパケットを検出するための再送信論理である。使用中再試行パケットはｒｅｔｒｙ＿ｐｋｔ待ち行列に待ち行列化し、実行済みパケットはトランザクション層に通知する。

【００３１】図２０は、実行済み（ａｃｋ＿ｄｏｎｅ）または使用中再試行を必要とするもの（ａｃｋ＿ｂｕｓｙ）として非ＳＳＯパケットを検出するための再送信論理である。使用中再試行パケットは再試行のためにＭＵＸにより送信し、実行済みパケットはトランザクション層に通知する。

【００３２】図２１は、リングレット内で未解決のＳＳ＊Ｏパケットが一切ない場合に使用中ループを設定し、再試行送信ブロックによりｒｅｔｒｙ＿ｐｋｔ待ち行列からパケットを選択するようにＭＵＸを設定することにより使用中再試行ループを制御するための再試行送信論理である。

デシリアライザ
８／１０デコード／ＣＲＣ検査
ｒｅｃ＿ｐｋｔレジスタ
ａｃｋデコード論理
ａｃｋ待ち行列

【００３３】図２２は、使用中再試行ループを開始するか、応答待ち行列から応答パケットを選択するか、または要求待ち行列から要求パケットを選択するための条件を検出するためのＭＵＸ論理である。

【００３４】図２３は、要求待ち行列から要求を選択するようにＭＵＸを制御するためのＭＵＸ論理である。

【００３５】図２４は、応答待ち行列から応答を選択するようにＭＵＸを制御するためのＭＵＸ論理である。

【００３６】図２５は、制御をフォーマットし、シリアライザへのパケットの送信をゲートするためのＣＲＣ生成論理である。

【００３７】図２６は、より大比較（．ＧＴ．と示す）とより小比較（．ＬＴ．と示す）を実行するための専用モジュロ６４比較論理である。

【００３８】図２９～３９の受信ノード用の流れ図の要約は以下に示す。また、図２７の受信ノード論理も参照する。

【００３９】この開示の拡張を理解するために、まず基礎となる論理を理解しなければならない。図２７では、ＳＳＯ順序付け拡張機能がない場合の受信ノードの基本ブロックは、以下に示す削除および変更を伴うすべてのブロックを含むものと思われる。

－ｓｅｑＴａｂｌｅ２６－３０を削除する
－ｓｅｑ　ｒｅｇｉｓｔｅｒ２６－４０を削除する
－ｓｅｑ　ｕｐｄａｔｅ２６－１２０を削除する

その他のブロックは、ＳＳＯサポートを伴わない受信ノードの動作にとって依然として基本的なものである。

【００４０】受信ノードと送信ノードは、共通の名前が付いた共通の構成要素をいくつか備えている。このような構成要素は以下の通りである。

送信ノード	受信ノード
７．２４０	２６．８０
７．２３０	２６．７０
７．２２０	２６．６０
７．１４０	２６．１４０
７．１３０	２６．１５０

ＳＳＯ順序付け論理の役割は、非ＳＳＯパケットに関する基本的なパケットの流れとの比較から理解することができる。図１０から始まり、ブロック９．１０、９．２０、９．３０では、ＣＲＣ＿ｉｎｉｔ＿ｅｒｒ、ＣＲＣ＿ｅｒｒ、ＣＲＣ＿ａｃｋ＿ｃｈｋからなるＣＲＣエラー処理制御順序が活動状態である場合に通常のパケット処理を禁止する。すべてのパケットが非ＳＳＯである場合、パケットを非ＳＳＯとして識別するブロック９．５０でパケット制御をブロック１１．１０に送る。

【００４１】非ＳＳＯパケットの流れ
上記に示唆したように、本システムは、非ＳＳＯ環境または完全ＳＳＯ環境のいずれかで使用することができる。

【００４２】基本的な（非ＳＳＯ）パケットの流れの説明ならびにＳＳＯサポートのために追加された図面の概要については以下の通りである。

【００４３】基本的なパケットの流れ、非ＳＳＯパケット

【００４４】図１２は、基本的な非ＳＳＯエラー検出論理を示しているが、肯定応答がパケットと一致するかどうかを確認するためにｓｅｎｄ＿ｐｋｔ待ち行列のヘッド（図８および図４０を参照）をａｃｋ待ち行列のヘッドと比較するものである。（比較はブロック１１．１０～１１．４０で行われる。）パケットにエラーがある場合、ブロック１１．５０は状態「ｅｒｒ＿ｒｅｔ」を設定し、これをエラー・パケットとして通知する。エラーがない場合、ブロック１１．６０は「使用中」状態と「実行済み」状態を（ａｃｋ待ち行列９．１３０のヘッドで）パケットの肯定応答から転送する。どちらの場合もパケットはｎｏｎ＿ｏｒｄｅｒ＿ｒｅｇに書き込まれる。

【００４５】ブロック９．１０には、非順序付けパケット用の再送信論理が記載されているが、ここではｎｏｎ＿ｏｒｄｅｒ＿ｒｅｇの有効性がテストされる。有効である場合、パケットはブロック９．２０で「実行済み」状況があるかどうかがテストされ、９．５０で「実行済み」状況がトランザクション層に送信される。パケットが「使用中」であるかまたは「ｅｒｒ＿ｒｅｔ」状態が設定されている場合、ブロック９．６０でｒｅｓｅｎｄ＄のために４：１＿ｍｕｘが設定され、パケットはｓｅｎｄ＿ｐｋｔレジスタに転送される。最後に、このブロックでは非ＳＳＯパケット用の順序カウントが増分され、この再試行パケットに割り当てられる。

【００４６】図２５では、ｓｅｎｄ＿ｐｋｔレジスタがブロック２４．１０で有効性がテストされ、次にブロック２４．３０でＣＲＣ生成に転送される。ｐＬａｂｅｌフィールドは、これが非ＳＳＯパケットであるという事実と、同じブロック２４．３０内の割当て済み「ｓｅｑ」フィールドによって定義される。また、このパケットは、もう一度ｓｅｎｄ＿ｐｋｔ待ち行列内に待ち行列化され、肯定応答パケットを待つ。

【００４７】流れ図はパケットの到着から始まるが、パケットは何らかの優先順位プロセスによって選択されたものであると想定する。非ＳＳＯシステムでは、この優先順位は本発明のシステムのものと同様になる可能性があるが、「使用中」および「ｒｅｔｒｙ＿ｐｋｔ待ち行列」への参照はすべて除去されるはずである。非ＳＳＯシステム内のそれぞれの待ち行列からの要求または応答パケットの選択は図２３および図２４に示すものと同様になるはずである。

【００４８】ＳＳＯサポートのための論理追加

図１０～１６の流れ図に示す残りのａｃｋ論理は、ＳＳＯ順序付けブロックを処理するための拡張機能を示している。

【００４９】ただし、ブロック１２．２０、１３．２０、１４．２０、１５．２０は、ｓｅｎｄ＿ｐｋｔ待ち行列内の非ＳＳＯ送信コマンドも識別し（図４０を参照）、非ＳＳＯパケット処理ブロック１１．１０へ分岐することに留意されたい。しかし、このようなブロックは、ＳＳＯと非ＳＳＯの両方のコマンドの混合プログラミング環境をサポートする機能を備えたシステム内の非ＳＳＯパケットの処理を包含する。すなわち、これらのブロックは、基本的な非ＳＳＯ専用環境の一部ではないのである。

【００５０】同様に、再送信論理ブロック内の非ＳＳＯパケットの処理についても、図１０の単一ページに示す。図１７～２０に示す再送信論理はＳＳＯサポートのために追加されたものである。再試行処理論理である図２１も、ＳＳＯ処理論理に固有のものである。

【００５１】使用中ループ（２１．９０に設定されている）を開始するための図２２に示すＭＵＸ論理は、ＳＳＯサポートを備えた実施態様に固有のものである。他の論理は要求または応答パケットを選択するものであり、ＳＳＯと非ＳＳＯのどちらの実施態様の場合にも同様のものになるはずである。図２３および図２４に示す要求または応答パケットを開始するためのＭＵＸ論理は、ＳＳＯと非ＳＳＯのどちらの実施態様の場合にも同様のものになるはずである。

【００５２】最後に、図２５のＣＲＣ生成論理は、２４．４０で「ｎｏ＿ｘｍｉｔ」状態ビットをテストし、パケットを逐次化論理に転送すべきかどうかを判定する。このようなゲートはＳＳＯサポートのために固有のものである。モジュロ６４比較をサポートする論理も、ＳＳＯサポートのために明確に必要なものである。

【００５３】図２７のブロック図の概要

図２７は、図８および図４０のように、受信ノード（送信ノードは図７）の潜在的なブロック図であり、ＳＳＯメカニズムの概念を実施する方法を示すために使用するものである。これは、このメカニズムの概念を実施する受信（送信）ノードの数多くの可能な実例の１つにすぎないことが分かっている。

【００５４】図２７内のパケットの流れは基本的に以下の通りである。－パケットは、リングレット２６．９０からデシリアライザ２６．８０を通って受信ノードに入る。ＣＲＣ検査と８／１０デコード論理２６．７０は非パケット記号を取り除き、パケットを３２ビットのｒｅｃ＿ｐｋｔレジスタ２６．６０にロードする。

－パケットの先頭が検出され、このレジスタ内で初期パケットデコードが行われる。受信ノードが使用するパケットは以下のものを含む。

－このノードにアドレス指定された送信パケット（要求および応答）

－他のノードにアドレス指定された送信パケット（要求および応答）

－他のノードにアドレス指定された肯定応答パケット

このノードにアドレス指定された送信パケットには、ｒｅｑｕｅｓｔ＿ｉｎ待ち行列（要求パケット）またはｒｅｓｐｏｎｓｅ＿ｉｎ待ち行列（応答パケット）にそれ

ぞれ待ち行列化するためにマークが付けられる。（注：待ち行列の使用）このノードへの肯定応答はデコードされ、ａｃｋ待ち行列２６．１５０に書き込まれる。

－受信ノード・パケットはｔｅｓｔ＿ｐｋｔノードに書き込まれる。パケットの先頭がｔｅｓｔ＿ｐｋｔレジスタ内にある場合、ｐｒｏｄｕｃｅｒＩｄを含む３２ビットがｒｅｃ＿ｐｋｔレジスタ内に入ってる。順序フィールドおよび受入れフィールドを順序付けするなど、このフィールドは受信ノード状態用のｓｅｃＴａｂｌｅアレイにインデックスを付ける。

－次のサイクルでは、ｓｅｑＴａｂｌｅ［ｐｒｏｄｕｃｅｒＩｄ］の内容がｓｅｑレジスタ２６．４０に読み込まれ、ｐＬａｂｅｌフィールドとｐｒｏｄｕｃｅｒＩｄフィールドはｔｅｓｔ＿ｐｋｔレジスタ２６．５０内に入る。これに対応するフィールドは、受信ノード内のすべてのパケット（送信パケットおよび肯定応答）について以下のブロックで比較される。ただし、パケット処理のタイミングは以下のようにパケットのタイプによって決まる。

－ａｃｐｔ論理２６．１３０は、基本的に、このノードにアドレス指定された要求および応答パケットが受け入れられるかどうかを判定する。ｓｅｑＴａｂｌｅおよびＣＲＣエラー検査の両方が検討される。

－ｓｅｑ更新論理２６．１２０は、パケットの末尾でｓｅｑＴａｂｌｅアレイ状態が更新されるかどうかと、その更新方法を判定する。

－ａｃｋ　ｇｅｎ論理２６．１１０は、ａｃｋ実行済み、使用中、またはエラーのうち、どの種類（ＣＲＣエラー検出時に禁止されるもの）の肯定応答が生成されるかを判定する。

－パケットの残りの部分が処理され（送信パケットの場合）、送信パケット・データはｒｅｑｕｅｓｔ＿ｉｎ（２６．１０）またはｒｅｓｐｏｎｓｅ＿ｉｎ（２６．２０）待ち行列内に待ち行列化される。ただし、待ち行列空間が使用可能である場合に限る。パケットが終了すると、ＣＲＣ検査は送信ノードのための動作を完了する。

【００５５】図２８Ａおよび図２８Ｂは、受信パケットと肯定応答のタイミング図をそれぞれ示している。受信ノード論理については、図２９以降に示し、以下に説明する。

【００５６】図２９では、ｒｅｃ＿ｐｋｔ＿ｒｅｇ２６．６０内のヘッダ・フィールドから送信パケット（または「コマンド」）または肯定応答パケットの先頭を検出する（図２７を参照）。送信パケットの場合、このノードにアドレス指定された要求または応答のいずれかにより、ｒｅｑｕｅｓｔ＿ｉｎ待ち行列２６．１０とｒｅｓｐｏｎｓｅ＿ｉｎ待ち行列２６．２０という対応する入力待ち行列内に待ち行列空間が存在するかどうかをテストし、それに応じて制御ビットが設定される。

【００５７】図３０では、ｓｅｑＴａｂｌｅアレイ２６．３０をｓｅｑ＿ｒｅｇ２６．４０に読み込む。ｒｅｑｕｅｓｔ＿ｉｎ待ち行列２６．１０またはｒｅｓｐｏｎｓｅ＿ｉｎ待ち行列２６．２０内のパケットの妥当性検査を禁止するための条件の有無をテストする。

【００５８】図３１では、待ち行列空間が使用可能であることを制御ビットが示す場合、このノードにアドレス指定されたパケットの残りの部分を対応する待ち行列、すなわち、ｒｅｑｕｅｓｔ＿ｉｎ２６．１０またはｒｅｓｐｏｎｓｅ＿ｉｎ２６．２０内に仮に待ち行列化する。制御フィールドを増分するかまたは制御ビットをリセットするための条件が与えられる。

【００５９】図３２では、このノードにアドレス指定された送信パケットの末尾で、ＣＲＣ検査に応じて、送信ノードへ肯定応答パケットを生成し、ｒｅｑｕｅｓｔ＿ｉｎ待ち行列２６．１０またはｒｅｓｐｏｎｓｅ＿ｉｎ待ち行列２６．２０内のパケットを妥当性検査する。

【００６０】図３３では、ＣＲＣエラー検査に応じて、図３２での肯定応答生成に使用するために仮の肯定応答条件（実行済み、使用中、またはエラー）を生成する。

【００６１】図３４では、有効順序を備えたＳＳＯパケット用の仮の肯定応答生成（実行済み、使用中、またはエラー）を続行する。

【００６２】図３５では、パケットの終了時にｓｅｑＴａｂｌｅアレイ用の更新条件を仮に設定するかまたは更新を禁止するための条件の有無をテストする。更新は、有効ＣＲＣ検査によって決まる。

【００６３】図３６では、パケットが有効順序で順序付けされたＳＳＯである場合に仮の順序更新生成を続行する。

【００６４】図３７では、送信パケットの終了時に、２６．７０内に有効ＣＲＣがあるかどうかをテストする。ＣＲＣが有効であり、ｓｅｑＴａｂｌｅ書込みが禁止されていない場合、ｓｅｑＴａｂｌｅを更新する。

【００６５】図３８では、他のノードにアドレス指定された肯定応答パケット用のｓｅｑＴａｂｌｅ更新フィールドを生成する。

【００６６】図３９では、パケットが有効順序で順序付けされたＳＳＯである場合に他のノードにアドレス指定された肯定応答パケット用のｓｅｑＴａｂｌｅ更新フィールドの生成を続行する。

【００６７】図４０以降の例は、本発明のシステムの動作例を実証するものであり、これらの図に示す状態およびアクションは上記の図の論理に適合するものである。図４０以降は、レジスタおよび待ち行列間のパケットの転送ごとの作成側ノードの状態をステップごとに示し、これらの図を検査することにより、本システムが使用中再試行ループおよび挙動不良のノードに関連してエラーが発生した場合のＳＳＯ非アイデンポテントコマンドに対応していることが分かる。

【００６８】各図の論理図／流れ図を見ると、様々な状態が示されている。これらについては以下に定義する。＊

状態	解釈
i	ｉｎ＿ｐｒｏｃ
b	使用中（変形：ａｃｋ＿ｂｕｓｙ、使用中ループ再試行）
r	ＣＲＣループ内の再試行
ｂ１	使用中再試行ループ内の第１のパケット
ｒ１	ＣＲＣ再試行ループ内の第１のパケット
x	ａｃｋまたは使用中のいずれか
d	ａｃｋ＿ｄｏｎｅ
ＢＲＣ	使用中再送信ループ
ＣＲＣ	ＣＲＣエラー再送信ループ
ＳＲＬ	ｓｅｎｄ＿ｐｋｔ再循環ループ

＊さらに、本明細書に付属の付録Ａには、現行設計に関連して使用する用語集および変数が記載されている。

【００６９】１３９４．２規格案を考慮した本発明の好ましい実施形態の説明

ＩＥＥＥ　Ｐ１３９４．２に基づくリングレット・トポロジで完全パイプライン化されたパケット開始をサポートするためのメカニズム案の概要および要約を以下に示す。この場合、パイプライン化パケットは、強力順次順序付け（ＳＳＯ）と非アイデンポテント・コマンドのサポートを必要とするアドレス空間からの送信パケットおよび応答パケットを含むことができる。このメカニズムは、トポロジ内のすべての通信リンクが順にＳＳＯ順序付けを維持することを確認することによって、任意のＰ１３９４．２スイッチ・トポロジにより拡張することができる。

【００７０】このメカニズム案は、非使用中非エラー・パケット送信のために完全パイプライン化動作モードでＳＳＯおよび非アイデンポテントコマンドをサポートし、効率のよいハードウェア再試行メカニズムによりＣＲＣエラーと受信側使用中条件の両方を許容する。これは、再試行した使用中パケットと新しいパケットとを混合するために様々な再送信方針をサポートする。ＣＲＣエラー・パケットはプログラム可能な限界まで再試行することができるので、これは「訂正可能エラー」の類のエラー処理方針を使用可能にする。この機能により、ＣＲＣエラー・パケットの正常な再試行動作を報告するノード用の「予防保守」機能が可能になる。

【００７１】また、ブリッジ内でＳＳＯの非アイデンポテント機能を使用すると、より大きい（＞６４Ｂ）パケット・フォーマットを有する他のプロトコルとＩＥＥＥ　Ｐ１３９４．２との間のブリッジの設計が大幅に単純化されることも留意されたい。たとえば、ＩＥＥＥ１３９４－１９９５ノードからの２ＫＢブロック分の書込みコマンドは、その終了時に受信側ノードから単一応答パケットを予期するが、一連の移動コマンド（応答なし）と終了書込みとして実施することができ、ＩＥＥＥ１３９４－１９９５パケットの動作がブリッジ上で保持されることを保証する。

【００７２】ＳＳＯ順序付けのコストは、リングレット上でサポートされるノード当たり２バイト分の状態と、状態マシン、非ＳＳＯパケット用の個別のレジスタ、ＭＵＸ、６ビットの比較器である。流れの説明は、より複雑なアレイ構造ではなく、単純なＦＩＦＯ待ち行列およびレジスタに基づいて示す。リングレット用の最大構成は６３ノードなので、これにより、最大１２８ＢまでＳＳＯ（ならびにポインタと状態マシン）をサポートするために必要な状態が制限される。サポートする構成がより小さい場合、その構成の「プロファイル」は、サポートするリングレット・ノード・カウントを切り詰めることにより、コストをさらに削減することができる。

【００７３】ＩＥＥＥ　Ｐ１３９４．２の簡単な概要

ＩＥＥＥ　Ｐ１３９４．２すなわちシリアル・エクスプレスは、ＩＥＥＥ１３９４－１９９５すなわちシリアル・バスをギガビット＋伝送レベルまで拡張するという提案である。基本的に、このプロトコルは、ＩＥＥＥ１３９４－１９９５の基本コマンド・セットをサポートしながら、１６Ｂのパケットおよび６４Ｂパケットにより待ち時間の低さとスループットの高さを強調するために転送プロトコルを再設計するものである。

【００７４】ＩＥＥＥ　Ｐ１３９４．２は、リングレット・トポロジを備えた挿入バッファ・アーキテクチャに基づくものである。これが意味するのは、各ノードが２本の単一方向ワイヤを含み、一方は着信パケットおよびアイドル（同期およびフロー制御用）向けであり、もう一方は発信パケットおよびアイドル向けであるということである。バイパス・バッファは、着信パケットおよびアイドルを発信ワイヤ上にそらし、おそらく遅延が発生する。単一の新しい送信または応答パケット（以下に定義する）は、そのバイパス・バッファが一杯ではない場合にノードによってリングレット内に挿入することができる。これは「挿入バッファ」アーキテクチャである。ノードが新しいパケットを挿入している間にバイパス・バッファに入るパケットは、新しいパケットが送信されるまで遅延される。パケット・サイズが小さいので（＜＝６４Ｂ）、このアーキテクチャが可能になるとともに効率のよいものになっている。

【００７５】パケット・フォーマットは、パケット経路指定用の１６ビットのフィールドと、４８ビットの拡張アドレスとを含む。各ノードは、別々の送信パスと受信パスとを備えた単一ケーブルによる全二重動作をサポートする。複数のノード（最高６３個まで）が相互接続されている場合、初期設定ソフトウェアにより冗長ループが切断され、これらのノードが単一リングレットとして構成される。複数のリングレット（接続部はわずか２カ所）は、１６Ｋノードからなる最大トポロジまでスイッチによって相互接続することができる。

【００７６】ＩＥＥＥ　Ｐ１３９４．２プロトコルは、２通りのアドレス・モード（有向およびマルチキャスト）と２通りのサービス・タイプ（非同期および等時性）に基づいて、４通りの動作モードをサポートする。ＳＳＯ順序付けは、非同期有向サービスに関して特定の利害があるものであり、（読取り、書込み、移動、ロック）諸動作を含むが、このメカニズム案はこのサービス・モードのみに限定されない。

【００７７】ＳＳＯ順序付け：発行

ＳＳＯ順序付けサポートがない場合、任意のノードから任意の宛先へのパケットには相互に関する制約が一切ない。ＣＲＣエラーと受信側使用中の両方に対処する場合、これは、２つのパケットが送信されたときとは異なる順序で宛先ノードに到着する可能性があることを意味する。

【００７８】この不確実性は、ネットワーク化環境を代表するものであり、様々なソフトウェアベースのプロトコルを使用して処理される。たとえば、従来のデータ・パケットを伴う割込み生成書込み要求を逐次化するためにＩＥＥＥ１３９４－１９９５で使用可能な信頼できるメカニズムの１つは、たとえば２ＫＢの大きいパケット用の書込みコマンド（宛先からの応答を要するもの）を送信し、次にそのデータに関する応答が戻ってきた後で割込みを生成するために書込みコマンドを送信することである。

【００７９】このようなメカニズムは、かなり小さい（＜＝６４Ｂ）パケットを備えたドメイン、すなわち、低待ち時間転送を強調するドメインに拡張されると、複雑かつ効率の悪いものになる可能性がある。順序付けとデータ転送完了をともに確立するためのソフトウェア・オーバヘッドにより、使用可能な計算サイクルが減少し、ユーザ・プロセスが、すなわちユーザ・プロセス待ち時間が直接増加する。

【００８０】Ｐ１３９４．２によるＳＳＯ拡張

ＳＳＯ拡張案は、リングレットの動作に関する上記の基本的な前提事項によって決まる。

【００８１】（非同期、有向）モードの各コマンドは、送信パケットを生成する発信側（送信側）ノードと、そのコマンドに応じて応答パケットを生成する（可能性のある）最終宛先ノードとによって実行される。「送信」パケットは、送信側のローカル・リングレット上の単一受信側ノードによって除去され、そのローカル受信側ノードから送信ノードに戻る「肯定応答」パケットに置き換えられる。ローカル受信側ノードは、送信パケット用の最終宛先である場合もあれば、おそらくスイッチによりその宛先にパケットを転送するエージェントである場合もある。コマンドのタイプに応じて、最終宛先ノードは「応答」パケットを送信することができ、その「応答」パケットはそのローカル・リングレット上の何らかのノードによって捕捉され、「応答」発生側に戻る「肯定応答」パケットによって置き換えられる。

【００８２】以下の説明では、そのノードが送信パケットを開始する送信側ノードであるか、応答パケットを開始する受信側ノードであるか、送信パケットまたは応答パケットを転送するブリッジまたはスイッチ・ノードであるかにかかわらず、パケットを開始したノードを示すために「作成側ノード」という用語を使用する場合もある。

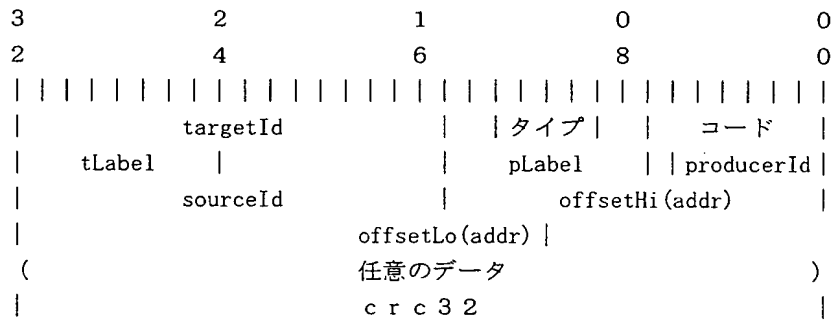
【００８３】送信パケットと受信パケット両方のターゲットは、グローバルな１６ビットの「ｔａｒｇｅｔＩｄ」フィールドによって識別される。エラーがない場合、リングレット上の何らかの固有のノード、おそらく、ブリッジまたはスイッチ・ノードは、ｔａｒｇｅｔＩｄアドレスを認識し、そのパケットを取り除く。「ｓｏｕｒｃｅＩｄ」フィールドは、送信ノードのグローバル・アドレスを明確に識別するものである。

【００８４】その他のフィールドは、ローカル・リングレット肯定応答を返すためにパケット内に含まれ、ＳＳＯ順序付け機能強化案にとって基本的なものである。このようなものとしては、＊＊そのローカル・リングレット上で＊＊このパケットを発信したｎｏｄｅＩｄを識別する（６ビットの）「ｐｒｏｄｕｃｅｒＩｄ」フィールドと、このパケットを明確に識別するために「ｐｒｏｄｕｃｅｒＩｄ」ノードによって割り当てられる（８ビットの）「ｐＬａｂｅｌ」フィールドがある。ｐｒｏｄｕｃｅｒＩｄフィールドとｐＬａｂｅｌフィールドはどちらも、ローカル・リングレット内に限定された意味を有し、たとえば、パケットがスイッチにより転送される場合に再割当てが行われる。同様に、「ｐｒｏｄｕｃｅｒＩｄ」フィールドと「ｐＬａｂｅｌ」フィールドは、宛先ノードが応答パケットを送信するときにその宛先ノードによる再割当ても行われる。

【００８５】以下に記述するその他のフィールドとしては、「タイプ」フィールド（コマンドおよび肯定応答のタイプを区別する）、「コード」フィールド（コマンド識別）、ｓｏｕｒｃｅＩｄノードに対してその応答を明確に識別するために応答パケットに入れて返されるグローバルな「ｔＬａｂｅｌ」フィールド、そのパケット用のヘッダとデータの両方を含む３２ビットのＣＲＣコードであるｃｒｃ３２フィールドがある。

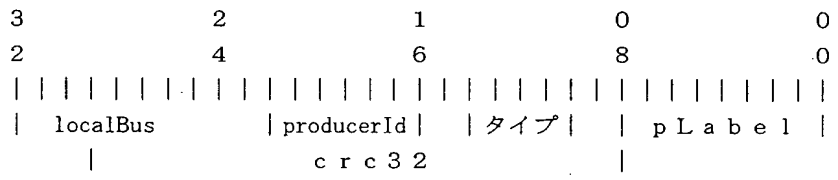
21　　22

（最初に送信されるもの）



【００８６】受信側ノードによって生成された肯定応答パケットは、送信または応答パケットから一部のフィールドをエコーして「作成側ノード」に戻すので、それは＊以下のように肯定応答中のパケットを明確に認識することができる。

（最初に送信されるもの）



【００８７】肯定応答アドレスを作成するために、元のパケットの「ｔａｒｇｅｔＩｄ」アドレス・フィールドは、「ｌｏｃａｌＢｕｓ」識別子（通常はすべて１）と元のパケットからの６ビットのｐｒｏｄｕｃｅｒＩｄフィールドによって置き換えられる。ｐＬａｂｅｌフィールドは「作成側ノード」に対して元のパケットを明確に識別するものである。「タイプ」フィールドは肯定応答のタイプをコード化するものである。肯定応答パケット用のｃｒｃ３２フィールドは、ここに記載するＳＳＯメカニズムのために普遍性を失わずに他の何らかのエラー検出コードによって置き換えることができる。

【００８８】ローカル・リングレット伝送が単一方向であり、いかなるノードもバイパスしないという想定は、本発明の提案にとって基本的なものである。すなわち、１つのノードが送信パケットまたは応答パケットのいずれかを送信すると、サブリング上のすべてのノードは、そのノードのバイパス・バッファを介してそのパケットの流れのためにそのパケットまたは肯定応答のいずれかを検出することになる。

【００８９】この想定は、Ｐ１３９４．２リングレットの基本動作の基礎となっている（ただし、Ｐ１３９４．２オプション案では、「ショート・カット」経路指定機能がＳＳＯ順序付けをサポートしないはずである）。この結果、伝送されるすべての送信応答パケットに関して、リングレット上の各ノードは、送信／応答パケットまたはその肯定応答のいずれかのためにｐｒｏｄｕｃｅｒＩｄフィールドとｐＬａｂｅｌフィールドの両方を監視することができる。

【００９０】Ｐ１３９４．２プロトコルは、応答を必要とする分割応答トランザクションをサポートする。（非同期、有向）サービス用のパケットには、以下のように※パケット配達を禁止する可能性のある２通りの条件がある。パケットまたはその肯定応答では、ＣＲＣエラーまたはその他の検出可能なエラーが発生する可能性がある。パケットは、宛先ノードで「使用中」の受信側によって拒否される可能性がある。

【００９１】ＣＲＣエラーの場合、信頼できる唯一の検出は発信側ノードで行われ、２回リングレット循環する間に予定の肯定応答が検出されない場合にエラーを検出することになる。（ローカル・ノード・タイムアウト検出を支援するためにリングレット循環ごとに補足される「スクラバ」ノードによって１つのビットが循環する。）使用するタイムアウト検出の正確な方法は、ここに記載するＳＳＯ順序付けサポート・メカニズムの動作にとって基本的なものではない。

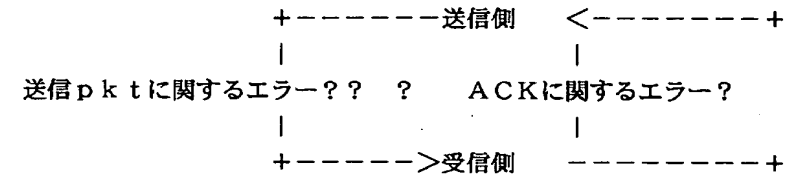
【００９２】受信側「使用中」の場合、受信側ノードは（非同期、有向）サービスの送信または応答パケットの代わりにＡＣＫ＿ｂｕｓｙ肯定応答パケットを使用する。

【００９３】システムの概要：ノードにおける既知の良好情報

ｐＬａｂｅｌフィールドとｐｒｏｄｕｃｅｒＩｄフィールドはリングレット・トポロジでＳＳＯ順序付けメカニズムを実施するために使用できるという主張は

【００９４】この場合の基本概念は、各パケットはリングレットで、ＳＳＯプログラム空間でコマンドまたはデータを伝送すること、およびリングレット順序状態情報を転送することという２つの使い方が可能である。送信パケットの順序番号は送信側だけが把握し、最後の有効受入れパケットは受信側だけが把握している。図で示すと以下のようになる。

23　　24



【００９５】作成側ノード：
－リングレット・ノード内のＳＳＯ動作を制御するための送信／応答パケット・フィールド：　各ｐｒｏｄｕｃｅｒＩｄからの送信および応答パケットに関するｐＬａｂｅｌフィールドを使用して３つのサブフィールド（以下に詳述する）伝達する。
pLabel.sso（１ビット）
pLabel.bzseq（１ビット）　／＊　非ＳＳＯ空間のために未使用　＊／
pLabel.seq（６ビット）
【００９６】「ｐＬａｂｅｌ．ｓｓｏ」ビットは、このパケット用のアドレス空間がＳＳＯ順序付き（ｓｓｏ＝１）であるかまたはそうではない（ｓｓｏ＝０）かを動的に定義する。「ｂｚｓｅｑ」すなわち使用中順序ビットは、ＳＳＯ順序付け空間内だけで意味を有し、それが送信／応答パケットに対して使用中再試行肯定応答を出さなければならないかどうかを受信側ノードが判定するのを支援するために使用する。６ビットの「ｓｅｑ」フィールドは、このｐｒｏｄｕｃｅｒＩｄから有効なパケット順序を識別するためにすべてのローカル・リングレット・ノードが使用する折返しモジュロ６４「カウンタ」である。
【００９７】リングレット上の各ｐｒｏｄｕｃｅｒＩｄノードと他のノードとの間のＳＳＯ順序付けを維持できるかどうかは、その送信および受信パケットを「測定」し、そのｐＬａｂｅｌフィールドを設定して他の各ノードが適切な状態情報を維持できるようにするｐｒｏｄｕｃｅｒＩｄによって決まる。その状態情報をｐｒｏｄｕｃｅｒＩｄからの着信ｐＬａｂｅｌフィールドと比較することにより、ＳＳＯ順序付けを維持するようにパケットを適切に処分することができる。
【００９８】「測定」では、作成側ノードが６ビットの「ｓｅｑ」フィールドに基づいてそのローカル・リングレット内で３２個程度の「未解決」パケットを有することができなければならない。「未解決」とは、「使用中」肯定応答を生成したこのノードからの最も早い先行パケットまたはそれに関して肯定応答を一切受信していない最も早い先行パケットのうち、いずれか大きい方へ返送するための現行パケットからのカウントを意味する。たとえば、宛先ノードで応答を待つなど、ローカル・リングレットを越えて未解決になりうるパケットの数については、いかなる限界も設計されていない。
【００９９】ＳＳＯ空間での通常動作では、新しい各送信／応答パケットまたは各再試行使用中パケットによってｐＬａｂｅｌ．ｓｅｑフィールドがモジュロ６４で１だけ増分される。ＣＲＣエラーが検出された場合、すべてのローカル・リングレットに関するＳＳＯ状態はまず、何らかのパケット（以下に示す）を再送信することによって再同期しなければならない。
【０１００】以下に使用する「ＣＲＣエラー」という用語は、一般に、ＣＲＣエラーだけでなく、送信ノードによって検出できるパケット（送信、応答、または肯定応答）用の非訂正エラー条件も含むものとして理解することができる。
【０１０１】ＣＲＣエラー・ループ・リセット：　作成側ノードでＣＲＣエラーが検出されると、リングレット内のすべてのノードでこのｐｒｏｄｕｃｅｒＩｄに関して維持されているＳＳＯ状態値が問題になる。このｐｒｏｄｕｃｅｒＩＤに関する各リングレット・ノードのＳＳＯ状態情報は、このｐｒｏｄｕｃｅｒＩｄまたは他の作成側ノードに関するＳＳＯプログラミング空間の変更なしにリセットしなければならない。
【０１０２】これを達成するために、このメカニズムの以下に示す２つの重要な特徴を使用する。
１．リングレット上の各受信側ノードは、ローカル・リングレット上の各送信ノード用のローカルｐｒｏｄｕｃｅｒＩｄによってインデックスが付けられたＳＳＯ状態のアレイを維持する。このＳＳＯ状態は、制御ビットと、２つの順序付けフィールドとを含む。これらのフィールドの一方は、この受信側にアドレス指定された現行リングレット・パケットが有効ＳＳＯ順序番号を有するかどうかを判定するために使用し、もう一方のフィールドは、ノードの受信バッファの状態とともに、そのパケットがノードによって受け入れられるかまたは拒否されるかと、どのタイプの肯定応答（実行済み、使用中、またはエラー）が返されるかを判定するために使用する。
２．ＣＲＣエラーを検出する各ｐｒｏｄｕｃｅｒＩｄは、エラー・パケットより前の最後の良好ＳＳＯ順序付きパケットから始まり、エラーの検出前にこのノードから送信された最後のパケットまで至る、一連のＳＳＯ順序付きパケットを再試行する能力を有する。リングレット内で有効順序ベース・カウントを確立するために、ＣＲＣエラー・ループを開始するように（必要であれば）既知の良好ＳＳＯ順序付きパケットを再試行する。次に、ＣＲＣエラー前の最後のパケットまで残りのパケットが再送信される。この再試行順序でエラーが検出された場合、それがエラーなしで正常に行われるか、またはエラー再試行ループが正常終了なしに完了するまで、ループを反復する。このような場合、よりレベルの高いプロトコルがエラーを処理することができる。この順序が

エラーなしで完了すると、リングレット内のすべてのノードは、そのＳＳＯ状態が一貫した既知の良好値に復元されることになる。

【０１０３】受信側ノード：受信側ノードは、そのＳＳＯ状態情報が正しい場合に、任意のｐｒｏｄｕｃｅｒＩｄから有効パケット順序を検出することができる。この状態は、リセット時にｐｒｏｄｕｃｅｒＩｄによって初期設定され、ｐｒｏｄｕｃｅｒＩＤによってＣＲＣエラーが検出された場合に再初期設定される。

【０１０４】受信側ノードは、ＣＲＣエラー送信パケットまたは肯定応答を検出する場合もあれば、検出しない場合もある。検出した場合でも、パケットの内容が信用できないので、ノード状態情報は一切更新することができない。タイムアウトを検出し、ＣＲＣエラー・パケットを再送信するのは、送信ノードの責任である。

【０１０５】しかし、受信側ノードは、何らかのｐｒｏｄｕｃｅｒＩＤからＣＲＣエラー・パケットに続く有効順序付きパケットの順不同伝送を検出することができる。このような順不同パケットのそれぞれは、ＡＣＫ＿ｅｒｒｏｒとして肯定応答する間に受信側で検出し拒否しなければならない。このパケットを送信するノードはパケットにマークを付け、そのエラー・ループでそれを再試行する。

＊【０１０６】パケットは、受信側ノードが非同期動作で拒否されるか、またはそのパケットが前に受け入れられたパケットの再伝送である場合にＳＳＯ順序付けサポートによって拒否される場合がある。ＳＳＯ順序付けモードでは、所与のｐｒｏｄｕｃｅｒＩｄからの第１のパケットが使用中として拒否されると、第１の使用中パケットが再試行されるまで、このｐｒｏｄｕｃｅｒＩＤからのすべての後続順序付きパケットも拒否しなければならない。

【０１０７】ＳＳＯの場合の受信側の状態は、非使用中受信側バッファへの新しいパケットの受入れと、エラー回復のために再送信される前に受け入れられたパケットおよび「使用中」拒否後に送信された新しいパケットの拒否をサポートしなければならない。作成側に返される肯定応答パケットはエラーの影響を受けるので、受信側状態は、任意のパケットが再試行される場合にａｃｋ＿ｄｏｎｅおよびａｃｋ＿ｂｕｓｙパケットを複製できなければならない。

【０１０８】これを達成するため、各受信側ノードは、以下の６つのフィールドを含み、ｐｒｏｄｕｃｅｒＩｄによってインデックスが付けられた、ｓｅｑＴａｂｌｅという状態アレイを必要とする。

＊

```
seqTable.verif[producerId]　（１ビット）
seqTable.sso[producerId]　（１ビット）
seqTable.seq[producerId]　（６ビット）
seqTable.bzseq[producerId]　（１ビット）
seqTable.busy[producerId]　（１ビット）
seqTable.acpt[producerId]　（６ビット）
```

【０１０９】ＳＳＯ動作では、ｓｅｑＴａｂｌｅフィールドを以下のように使用する。「ｖｅｒｉｆ」フィールドは、このｓｅｑＴａｂｌｅが初期設定されたことと、ｐｒｏｄｕｃｅｒＩｄが検査済みソースであることの両方を示す。「ｓｓｏ」フィールドは、ＳＳＯ順序付けをサポートするためにこのｐｒｏｄｕｃｅｒＩｄノードの（静的）機能を定義する。（比較によれば、ｐＬａｂｅｌ．ｓｓｏビットは、作成側ノード用のアドレス空間順序付けに応じて設定された動的ビットである。）「ｓｅｑ」フィールドは、受信側ノードによるパケット順序付けのためのｐＬａｂｅｌ「ｓｅｑ」フィールドと比較される。「ｂｚｓｅｑ」ビットは、第１の使用中パケットの再伝送を識別するためにＳＳＯパケット用の対応するｐＬａｂｅｌ「ｂｚｓｅｑ」ビットと比較される。「ｂｕｓｙ」ビットは、ＳＳＯパケットの第１の使用中肯定応答時にｐｒｏｄｕｃｅｒＩｄに設定され、受信側バッファに応じてこの使用中パケットが再試行されたときに（おそらく）リセットされる。最後に、「ａｃｐｔ」フィールドは、ＳＳＯパケットの受入れおよび拒否の場合と、「実行済み」または「使用中」として肯定応答すべきかどうかを判定する場合の両方に使用する。このような判定では、着信パケットの「ｓｅｑ」フィールドとの比較が必要である。

【０１１０】受信側ノードは、以下のような場合に所与のｐｒｏｄｕｃｅｒＩＤから有効な新しいＳＳＯ順序付けパケットを検出することができる。

```
pLabel.seq = seqTable.seq[producerId] + 1
```

【０１１１】比較が有効である場合、seqフィールドが更新される。

```
seqTable.seq[producerId] = pLabel.seq
```

【０１１２】その他の比較については、以下を参照されたい。

【０１１３】このｐｒｏｄｕｃｅｒＩｄがＣＲＣエラー再試行ループを初期設定することをこのノードが検出した場合、ｓｅｑＴａｂｌｅ．ｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］フィールドが現行のｐＬａｂｅｌ．ｓｅｑ値にリセットされる。これは、以下のようにこのｐｒｏｄｕｃｅｒＩｄからのパケットまたはこのｐｒｏｄｕｃｅｒＩｄへの肯定応答用のｓｅｑフィールドを比較することによって示される。

```
if (pLabel.seq <= seqTable.seq[producerId]) then
seqTable.seq[producerId] = pLabel.seq
```

【０１１４】ＣＲＣエラー・ループの初期設定中は、ｓｅｑＴａｂｌｅ．ａｃｐｔ[]フィールドではなく、ｓｅｑＴａｂｌｅ．ｓｅｑ値だけが変更される。パケットは拒否されるが、通常使用するのと同じ受入れ／拒否論理（以下に示す）を使用する。再試行したすべてのパケットを「実行済み」（前に受け入れられたことを意味する）または「使用中」として正しく肯定応答するためにｓｅｑＴａｂｌｅ状態情報を使用することが重要である。論理については以下に説明する。

```
accept : {  {pLabel. bzseq～＝seqTable
            . bzseq [producerId]}
            &{receiver__not__busy}
         |  {pLable. bzseq＝seqTable.
            bzseq [producerId]}
            &{seqTable. busy [produc
               erId] ＝0}
            &{receiver__not__busy}
            &{pLabel. seq＞seqTable.
               acpt [producerId]}}
reject : {  {receiver__busy}
         |  {pLabel. bzseq＝seqTabel.
            bzseq [producerId]}
            &{seqTable. busy [produc
               erId] ＝1}
         |  {pLabel. bzseq＝seqTabel.
            bzseq [producerId]}
            &{pLabel. seq＜＝seqTabel
               . acpt [producerId]}}
      ＝～accept
```

【０１１５】「ｓｅｑ」比較が有効である場合、パケットはこの比較に応じて受け入れられるかまたは拒否される。

【０１１６】あまり形式的ではないが、使用中再試行ループが開始され、受信側バッファが使用中ではない場合、または使用中再試行ループは開始されていないが、「使用中」状態がリセットされ、受信側バッファが使用中ではなく、作成側ノードがエラー再試行ループに入っていないと順序比較が示している場合、パケットは受け入れられる資格を有する。

【０１１７】「ｓｅｑ」比較が有効である場合、パケットはこの比較に応じて「実行済み」または「使用中」として肯定応答される。

```
ack__don : {  {pLabel. bzseq～＝seqTabl
              e. bzseq [producerId]}
              &{receiver__not__busy}
           |  {seqTable. busy [produce
              rId] ＝0}
              &{receiver__not__busy}
           |  {pLabel. seq＜seqTable. a
              cpt [producerId]}}
act__bzy : {  {pLabel. bzseq＝seqTabel
              . bzseq [producerId]}
              &{seqTable. busy [produ
                 cerId] ＝1}
              &{pLabe. seq＞＝seqTabel
                 . acpt [producerId]}
           |  {pLabel. seq＞＝seqTabel.
              acpt [producerId]}
              &{receiver__busy}
              ＝～ack__don
```

【０１１８】あまり形式的ではないが、使用中再試行ループが開始され、受信側が使用中ではない場合、または

「使用中」状態がリセットされ、受信側が使用中ではない場合、またはエラー再試行ループでは再試行中のパケットが第１の「使用中」パケットより前に送信されたことを順序比較が示している場合、パケットはａｃｋ＿ｄｏｎｅとして送信される。それらがａｃｋ＿ｄｏｎとして肯定応答されない場合、パケットはａｃｋ＿ｂｚｙとして肯定応答される（この場合も、有効な順序比較が前提事項となる）。

【０１１９】－受信側ノードが初めて所与のｐｒｏｄｕｃｅｒＩｄから使用中としてパケットを拒否する場合、（ｓｅｑＴａｂｌｅ．ｂｕｓｙ［ｐｒｏｄｕｃｅｒＩｄ］＝１）を設定することにより、その「使用中」状態を変更しなければならない。これは、「使用中」パケットの順序番号でｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドをフリーズするという影響を及ぼす。このフィールドは、第１の使用中パケットが再試行されるまでフリー＊ズしたままになる。

【０１２０】「ｂｕｓｙ」フィールドと「ａｃｐｔ」フィールドを設定するための条件は、使用中再試行ループが作成側ノードによって実行されるかどうかによって決まる。「ｓｅｑ」比較が有効である場合、使用中再試行ループ内の第１のパケットは以下の関係から検出することができる。

pLabel.bzseq ～＝ seqTable.bzseq[producerId]

ただし、「ｂｚｓｅｑ」フィールドは、エラー再試行ループを実行中に変更されないことに留意されたい。これは真実なので、使用中再試行ループのパケットにおいて以下の比較は必ず真になる。

{pLabel.seq ＞ seqTable.acpt[producerId]}

【０１２１】第１の再試行パケットが検出された場合、ｓｅｑＴａｂｌｅの各項目は以下のように設定される。

```
{  seqTabel.bzseq[producerId]
     =pLabel.bzseq
   seqTabel.busy[producerId]
     =receiver_busy
   seqTable.acpt[producerId]
     =pLabel.seq  }
```

【０１２２】ただし、第１の再試行パケットは、上記の比較基準により拒否され、「実行済み」または「使用中」として肯定応答されることに留意されたい。パケットが第１の再試行パケットではない場合、これは以下の※関係から検出される。

```
if {  {seqTable.busy[producerId]
       =0}
       &{pLabel.seq>seqTable.acpt[
          producerId]}}  then
        {seqTable.busy[producerId]
         =receiver_busy
         seqTable.acpt[producerId]
         =pLabel.seq
          {
else／＊ 前の使用中パケットまたはエラー再試行ループ ＊／
     {no  update
    }
```

pLabel.bzseq = seqTable.bzseq[producerId]

【０１２３】次に、ｓｅｑＴａｂｌｅの各項目は以下のようにパケット受入れによって決まる。

【０１２４】最後に、各ノードは、以下のような場合にｐｒｏｄｕｃｅｒＩｄから順序エラーを含むパケットを検出することができる。

pLabel.seq ＞ seqTable.seq[producerId] + 1

【０１２５】これは、このパケットには有効ＣＲＣがあるが一部の先行パケットにはなかった場合に発生する可能性がある。（カウント比較は、完全６ビットの算術比較ではない。以下の詳細を参照されたい。）順序エラーを含むパケットについては、ｓｅｑＴａｂｌｅの各項目は一切更新されない。

【０１２６】リングレットＳＳＯ状態の初期設定：

－作成側ノードは、実際のＳＳＯ送信／応答パケットを送信する前にそのリングレット上の各ノードでの状態情報が初期設定されることを確認しなければならない。

【０１２７】「承認作成側」という方針は、ｓｅｑＴａｂｌｅを初期設定するために提案されたｓｅｑＴａｂｌｅ状態ビットおよび書込み送信パケットによってサポートすることができる。以下の３つのステップが必要である。

１．ｓｅｑＴａｂｌｅ[]アレイは、電源がオンになるか、またはすべてのｐｒｏｄｕｃｅｒＩｄについてｓｅｑＴａｂｌｅ．ｖａｒｉｆ[]=０を指定してリセットされる。

２．非ＳＳＯアドレス空間内の（未指定）プロセスによ

り、ローカル・ノードはｐｒｏｄｕｃｅｒＩｄ案を検査し、それ自体のｓｅｑＴａｂｌｅ．ｖｅｒｉｆ［ｐｒｏｄｕｃｅｒＩｄ］＝１を書き込む。「ｖｅｒｉｆ」ビットは、ローカル・ノードによって書き込まれるだけである。

３．ｓｅｑＴａｂｌｅ．ｖｅｒｉｆ［ｐｒｏｄｕｃｅｒＩｄ］が活動化された状態で、ｐｕｒｏｄｕｃｅｒＩｄ＊はこのノードのｓｅｑＴａｂｌｅにアドレス指定された書込み（または移動）コマンドを送信する。パケットは、そのターゲット・アドレス（ｓｅｑＴａｂｌｅ）により初期設定パケットとして識別される。

【０１２８】ｓｅｑＴａｂｌｅに書き込まれる値は、以下のように初期設定パケットのデータから取られる。

```
if {seqTabel. verif [producerId]
    ==1} then
   {seqTable. sso [producerId] =1
    seqTable. seq [producerId]
    =[current" seq" from this produ
      cerId]
    seqTable. bzseq [producerId]
    =[current" bzseq" from this pro
       ducerId]
    seqTable. busy [producerId] =0
    seqTable. acpt [producerId]
    =[current" seq" from this produ
      cerId]
   }
```

【０１２９】リングレット内の各ノードは、別々に検査し初期設定しなければならない。

【０１３０】上記のような方針を使用すると、既存のリングからノードを動的に追加（または削除）すると同時に新たに電源オンしたリングを初期設定することができる。

【０１３１】上記のような方針は、選択的「承認ノード」交換を可能にすることができる。たとえば、ノードＡはノードＢと、ノードＣはノードＢと交換することができるが、ＡとＢがＳＳＯ交換されるのを防止することができる。

【０１３２】応用空間

このメカニズム案は、以下のものを含む汎用１３９４．２ネットワーク・トポロジで動作することができる。ＳＳＯ順序付けが可能なノードと、そうではない（静的機能）のノード、およびＳＳＯ順序付けされたかまたはＳＳＯ順序付けされていない（動的機能）複数のアドレス空間をサポートするノード。

【０１３３】ＩＥＥＥ　Ｐ１３９４．２内にはＳＳＯ順序付けされていない可能性のあるモード（等時性）が存在するので、動的機能は重要であり、非ＳＳＯ順序付け空間へのブリッジ（ＩＥＥＥ１３９４－１９９５など）をサポートしなければならないことに留意されたい。一般に、ＩＥＥＥ　Ｐ１３９４．２ノードへのソフトウェア・インタフェースは、潜在的に変動するプログラミング・モデルを備えた１組のアドレス空間であると推定される（必須ではない）。これらのモデルのうちの少なくとも１つは、非アイデンポテント・コマンドをサポートするＳＳＯモデルであると推定される。

【０１３４】ＳＳＯ順序付けメカニズムはＩＥＥＥ　Ｐ１３９４．２（非同期、有向）サービスに応用されるので、その応用については以下に記載する。これは、このメカニズムが他のモードには応用できないことを意味するわけではない。このメカニズムは、複数のアドレス空間からの任意のコマンド・ストリームをサポートし、ＳＳＯコマンドに他のプログラミング・モデルからのコマンドが散在できることを意味する。

【０１３５】ＳＳＯアドレス空間（またはＳＳＯドメイン）から宛先ノードへの終端間でＳＳＯ順序付けを維持する必要がある場合、ＳＳＯドメイン・コマンド用の宛先ノードがＳＳＯ順序付け可能でなければならず、スイッチまたはブリッジ内のすべての中間ノードがＳＳＯ順序付け可能でなければならないことは明らかである。

【０１３６】サポートに必要なＩＥＥＥ　Ｐ１３９４．２の変更

ＳＳＯサポート・メカニズム案は、文書化されていない任意の機能として変更なしに現行規格の０．６バージョンで実施することができる。しかし、このメカニズムを含む特許を出願すると、（少なくとも）任意の機能としてこのメカニズムをＩＥＥＥ　Ｐ１３９４．２の文書に追加することが計画される。

【０１３７】変更は不要であるが、明確にするために１つのコード化の追加が示唆され、費用効果の高いＩＥＥＥ１３９４－１９９５へのブリッジをサポートするためにもう１つのコード化の追加が必要である。

【０１３８】明確にするためのコード化の追加は、「タイプ」フィールドに「ＡＣＫ＿ｅｒｒｏｒ」肯定応答を追加し、予約コードを置き換えるものである。

33

Ｔｙｐｅ　ｆｉｅｌｄ：０　Ｎｏｒｍａｌ，　ｎｏｎ－ＳＳＯ
１　Ｅｘｔｅｎｄｅｄ，　ｎｏｎ－ＳＳＯ
２［ｒｅｓｅｒｖｅｄ］
３　［ｒｅｓｅｒｖｅｄ］
４　ＡＣＫ＿ｄｏｎｅ
５　ＡＣＫ＿ｂｕｓｙ
６　ＡＣＫ＿ｍｏｒｅ／*マルチキャスト・モード再試行が必要*／*７ＡＣＫ＿ｅｒｒｏｒ*／*ＳＳＯモードでの順序エラー*／

【０１３９】ブリッジのためのコード化の追加では、より長いパケット・タイプをサポートする相互接続によりその宛先に達する、開始、中間、終了の各６４Ｂパケットの輪郭を示すビットを指定する。たとえば、イーサネットまたは１３９４による２ＫＢパケットは、以下の３通りのタイプのパケットに分解することができる。

開始パケット：　おそらく部分ブロックであり、６４Ｂ境界上のその末尾に位置合せされる。

中間パケット：　必ず６４Ｂで、必ず位置合せされる。

終了パケット：　おそらく部分ブロックであり、６４Ｂ境界上のその先頭に位置合せされる。

【０１４０】ブリッジによりＰ１３９４．２リングレットに入る大きいパケットは、より小さいパケットに分解することができる（たとえば、１つの大きい書込みパケットは複数の６４Ｂの移動と１つの６４Ｂの書込みとに分解される）。しかし、たとえば、１３９４宛先ノードにブリッジするために、偶数のＳＳＯ順序付き６４Ｂのパケットから１つの大きいパケットを再組立てするには、大きいパケットの構成要素部分が容易に識別できなければならない。大きいパケット用に設計されたプロトコルで小さい６４Ｂのパケットを処理する場合、通常、効率が悪くなるので、この必要性が特に強くなる。開始／中間／終了コード化をＳＳＯ順序付け機能と組み合わせると、ブリッジは、配達のために非常に効率よくパケットを再組立てすることができる。

【０１４１】再試行の概要

非アイデンポテント・コマンドをサポートするＳＳＯ順序付きアドレス空間のために、受信側ノードでの「使用中」条件と、ローカル・リングレットで検出されるＣＲＣエラー条件という２通りのタイプの再試行条件を処理しなければならない。後者の場合、「ＣＲＣエラー」は、ソース・フィールドと宛先フィールドのいずれも信頼できるデータを表さないほどのパケット（送信パケット、応答パケット、またはそのいずれかへのＡＣＫ）の*

34

*破壊を意味するものと想定されている。「ＣＲＣエラー」という用語は、ＣＲＣエラー・コードのみに制限することを意味するものと解釈すべきではなく、より一般的には任意のエラー検出メカニズムを含むべきである。順序付きドメインの効率のよい動作のために、「使用中」再試行条件は比較的頻度が低いが、ＣＲＣエラー条件よりかなり頻度が高いと想定する。しかし、いずれの場合も、提案した方式の有効性は、「使用中」またはＣＲＣエラーのいずれかのケースの発生頻度に依存していない。

【０１４２】順序付きドメイン内の「使用中」再試行の場合、エラーがなければ、「使用中」と応答する特定のドメインへのパケットの再試行だけが必要である。提案した方式では、代替宛先ノードでの「使用中」条件を独立したものとして扱う。ノードＡが一連のパケットをノードＢとＣの両方に転送し、Ｂのみが「使用中」と応答する場合、ノードＢへのパケットだけが再試行される。

【０１４３】ＣＲＣエラー・パケット（送信／応答パケットまたはＡＣＫのいずれか）は最終的に送信側ノードで検出される。ＳＳＯ順序付きドメインでは、この検出は、「順序エラー」（タイムアウト前にＡＣＫ＿ｅｒｒ応答が検出される）またはリングレット・タイムアウト（肯定応答なしで２回循環する）のいずれかになる可能性がある。ＳＳＯ順序付きドメイン内の受信側ノードは、何らかのノードから順序エラーが検出された場合にＣＲＣエラーの発生後に各送信／応答パケットを拒否するためにＡＣＫ＿ｅｒｒｏｒを送信するという責任を負っているが、そうではない場合はエラー状態情報を維持する必要はない。そのパケットまたはＡＣＫのいずれかでＣＲＣエラーが発生したと送信側ノードが検出すると、そのノードは、最後の有効完了パケットの点から、エラーが検出され、パケット伝送が停止される前に送信された最後の未解決パケットまで、すべての未解決ＳＳＯ順序付きドメイン・パケットを再試行しなければならない。非ＳＳＯドメインでＣＲＣエラーを再試行するための方針については、ここでは指定しない。

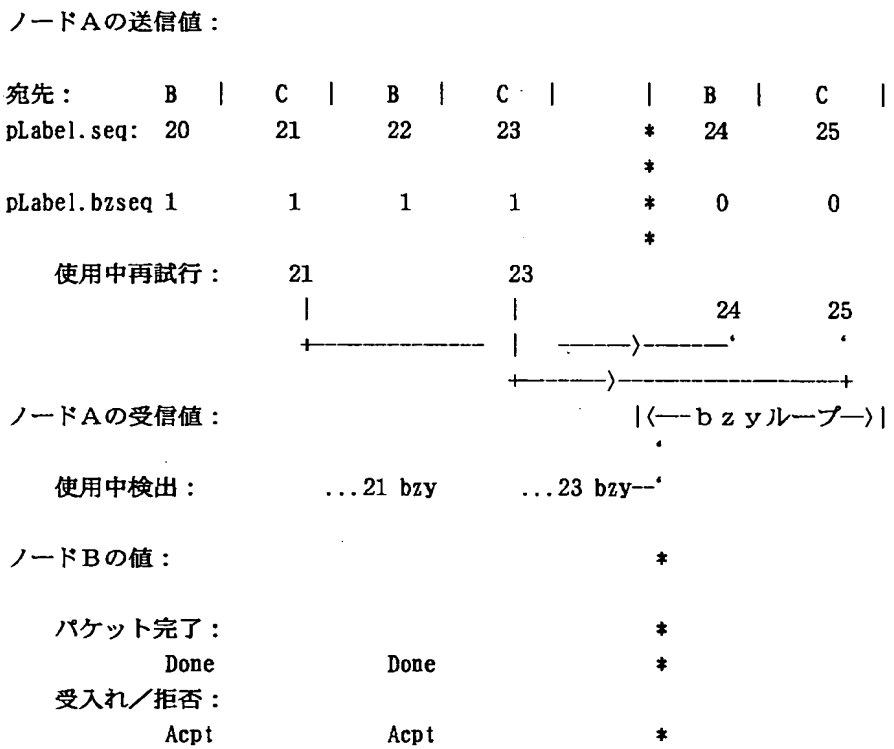
【０１４４】使用中再試行の例

このメカニズムがどのように機能するかをより適切に視覚化するために、詳細説明を示す前に例を使用してＳＳＯ順序付けサポート用のメカニズム案を例示する。

【０１４５】ケース１：　ノードＡからノードＣへの使用中再試行：

送信パケットの観点からのパケット順序付け：

35　　36



【０１４６】この例に関するいくつかの所見：パケットは送信ポートを介して生成されるが、使用中肯定応答は受信ポートで非同期に検出される。使用中再試行ループでは、第１の再試行使用中パケット（２４）を再伝送する前にすべての未解決先行トランザクションに肯定応答しなければならない。また、いつ使用中パケットを再試

行すべきかという判断は、新しいトランザクションの到着速度など、実施上の他の考慮事項によって決まる可能性がある。この例のパケット２３は、使用中ループがパケット２１および２３を再番号付けした（ｐＬａｂｅｌ．ｓｅｑ）パケット２４および２５として再送信する前に送信される最後のパケットになる。使用中のマークが付けられたパケット（この例では、パケット２１から始まるノードＣへのパケットのみ）だけが再試行される。完了したパケットを再試行する必要はない。新しい順序番号は、ノードＡによって再試行したパケットに割り当てられる。第１の「使用中」パケットがノードＣ（パケット２１）で肯定応答されると、同じかまたはそれ以上の順序番号がノードＣにアドレス指定されたすべてのパケット（この例ではパケット２３）は、第１の「使用中」パケットが再試行される（「ｂｚｓｅｑ」の変更によって示される）まで、「ａｃｋ＿ｂｕｓｙ」によって拒否しなければならない。エラー再試行ループでは、２１の前にパケットが再伝送される可能性がある。この場合、それが（間違って）再試行されるのを防止するため、そのパケットは「ａｃｋ＿ｄｏｎｅ」として拒否され肯定応答されるはずである。第１の「使用中」パケットに肯定応答する他のすべてのノードは同じように動作しなければならない（この例には示さない）。ＳＳＯ順序付けは、送信側ノードからその潜在的な受信側ノードのそれぞれへ、対単位で維持される。しかし、ノードＡがノードＢに対してＳＳＯ順序付きパケットを送信し、Ａがノードにもノードに順序付きパケットを送信する場合、結果的に、対単位のＳＳＯを正しくするために＊ノードＢとＣとの間の相対的な順序付けパケット到着を維持しなければならないというわけでは「ない」。このｐｒｏｄｕｃｅｒＩｄノードによる使用中再試行ループの開始は、ｐＬａｂｅｌ．ｂｚｓｅｑフィールドを（１から０に）補足することにより、すべてのリングレット・ノードに通知される。このスワップは、そのｓｅｑＴａｂｌｅ．ｂｚｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］値を着信ｐＬａｂｅｌ．ｂｚｓｅｑと比較することにより、各リングレット・ノードで検出することができる。両方の値が一致しない場合、ｓｅｑＴａｂｌｅ値は以下のように書き換えられる。

ｓｅｑＴａｂｌｅ．ｂｚｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］　＝　ｐＬａｂｅｌ．ｂｚｓｅｑ

また、再試行した使用中パケットが受信バッファに受け入れられた場合、「使用中」ビットは以下のようにリセットされる。

ｓｅｑＴａｂｌｅ．ｂｕｓｙ［ｐｒｏｄｕｃｅｒＩｄ］　＝　０

すべての他の（非受信側）ノードの場合、ｓｅｑＴａｂｌｅ．ｂｕｓｙ［ｐｒｏｄｕｃｅｒＩｄ］は０に設定される。

【０１４７】ＣＲＣエラー再試行の例

使用中肯定応答を行わないＣＲＣエラー再試行の２つの例を、使用中再試行ループ周辺で発生するＣＲＣエラーの一例とともに以下に示す。

【０１４８】ケース１：　ノードＣで検出されたノードＣへの送信パケット上のＣＲＣエラー送信パケットの観点からのパケット順序付け：

39 40

宛先：

```
     B  |    C  |    B  |    C  |    B  |    C  |    B  |    C
 pLabel.seq:
     20       21       22       23 *    20       21       22       23
  *
 pLabel.bzseq
     1        1        1        1  *     1        1        1        1
                                  *
 ＣＲＣエラー再試行：
     20
     |                                  20
     +----------------〉-------‘
ノードＡの受信値：                      |〈----------ループ----------〉|
                                        ‘
 使用中検出：          ...２１　エラー―‘
ノードＢの値：                          *
 パケット完了：                         *
    Done          *   Done                Done                Done
 受入れ／拒否：
    Acpt              Acpt              * Rej                 Rej
                                        *
 seqTable.seq[producerId]:              *
     19       20       21       22  *    23       20       21       22
 seqTable.bzseq[producerId]:            *
     1        1        1        1  *     1        1        1        1
 seqTable.busy[producerId]:             *
     0        0        0        0  *     0        0        0        0
 seqTable.acpt[producerId]:
              *
     19       20       21       22 *    23       20       21       22
                                  *
ノードＣの値：                          *
 Packet completion:                     *
            (CRC)                 Err *                Done                Done
 Accept/reject:
            (err)                 Rej *                Acpt                Acpt


                                  *
 seqTable.seq[producerId]:              *
     19       20       20       20 *    20       20       21       22
 seqTable.bzseq[producerId]:            *
     1        1        1        1  *     1        1        1        1
 seqTable.busy[producerId]:             *
     0        0        0        0  *     0        0        0        0
 seqTable.acpt[producerId]:             *
     19       20       20       20 *    20       20       21       22
```

【０１４９】この例に関するいくつかの注意事項：ＣＲＣエラーの検出は、送信ノードから非同期に受信ノード内で行われる。この例では、２１のＣＲＣエラーが検出される前にパケット２２および２３が伝送される。ＣＲ

41

Ｃエラーが発生しうる個所が多いので、様々なノードは様々なやり方でエラーを検出できる場合もあれば、検出できない場合もある。この例では、ノードＢの後でＣＲＣエラーが発生しているので、順序エラーは決して発生しない。これに対して、ノードＣはパケット２１を検出できず、後続のパケット／ａｃｋ２２および２３に順序エラーが発生していることを検出する。というのは、最後に記録されたその順序値が２０であるからである。ＣＲＣエラーが送信側で検出されると、各ローカル・リングレット・ノードのｓｅｑＴａｂｌｅ．ｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］フィールド内で順序カウントを（２０まで）再確立するために、そのＣＲＣエラー・ループは、エラー（パケット２０）の前に、必要であれば繰返し、まず最後の良好パケットを再伝送しなければならない。受信側ノードのｓｅｑＴａｂｌｅ．ａｃｐｔ［ｐｒｏｄｕｃｅｒＩｄ］フィールドとの比較に基づいて、ターゲットとは無関係に、このパケットは必ず拒否される。既知の有効パケット用の順序番号とともに、ｐｒｏｄｕｃｅｒＩｄもｐＬａｂｅｌに入れて現行のｂｚｓｅｑ値を送出し、すべてのｓｅｑＴａｂｌｅ．ｂｚｓｅｑ＊

42

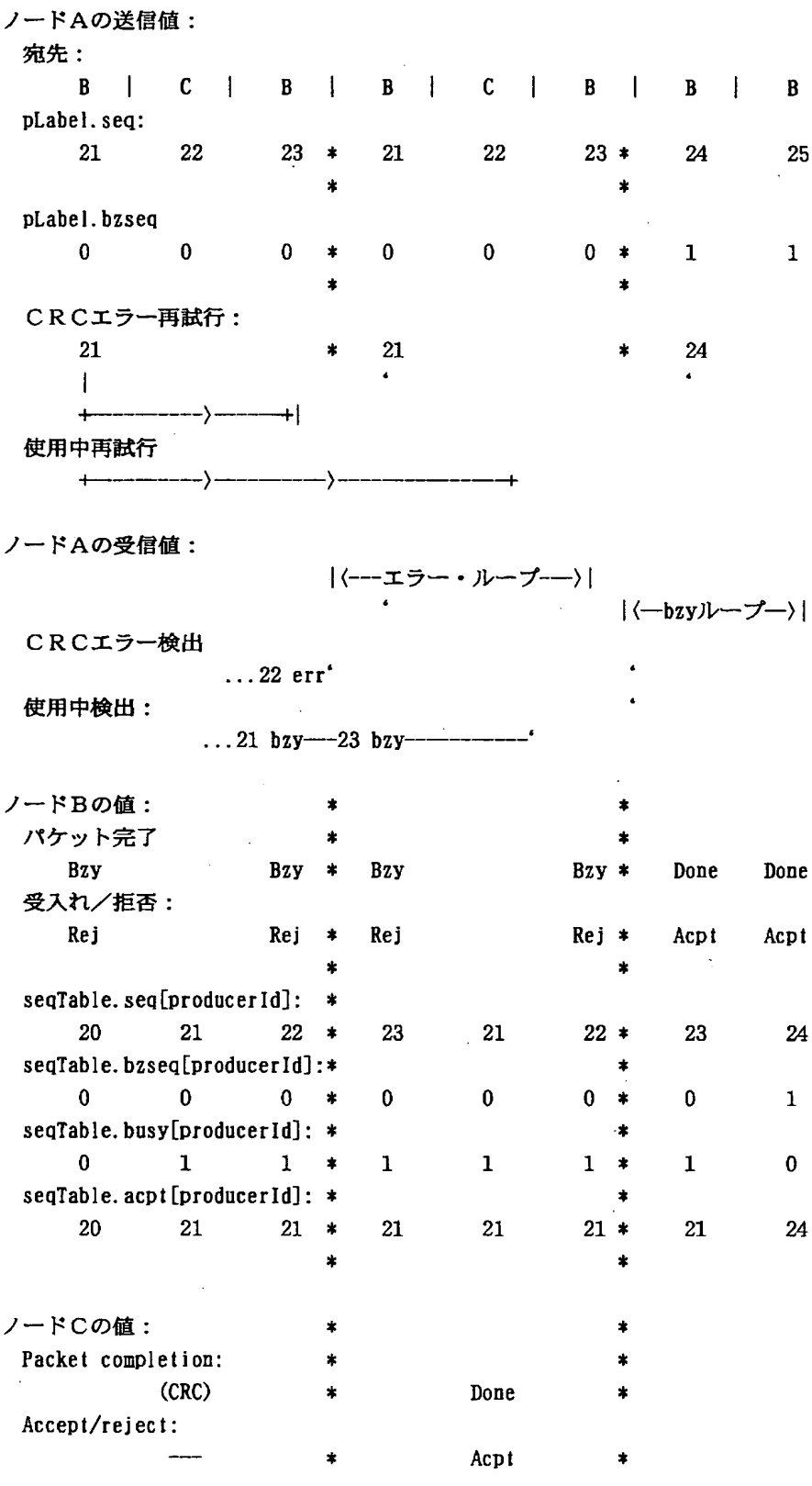
＊［ｐｒｏｄｕｃｅｒＩｄ］を既知の良好状態に設定する。ｓｅｑＴａｂｌｅ．ｂｚｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］値はリセットされるが、各ノードのｓｅｑＴａｂｌｅ．ｂｕｓｙ［］フィールドとｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドは保持しなければならない。これらのフィールドは、この宛先用の正しい応答ａｃｋ、実行済み、または使用中によってＣＲＣエラー再試行ループでパケットを受け入れるかまたは拒否するために必要な状態を表す。この例では、ノードＢは送信／応答パケット２２を拒否し、ノードＣはパケット２１と２３の両方を新しいパケットとして受け入れる。送信側は、新しいパケットを停止させた点まで、この場合はパケット２３まで伝送されたすべてのパケットを再送信しなければならない。エラーがない場合、ノードＢとＣは順序番号を追跡する。

【０１５０】ケース２：　ノードＡで検出されたノードＣからの肯定応答パケット上のＣＲＣエラー

送信パケットの観点からのパケット順序付け：

送信パケットの観点からのパケット順序付け：

43 44


ノードＡの送信値：
宛先：
B | C | B | B | C | B | B | B
pLabel.seq:
21 22 23 * 21 22 23 * 24 25
pLabel.bzseq
0 0 0 * 0 0 0 * 1 1
ＣＲＣエラー再試行：
21 * 21 * 24
使用中再試行
ノードＡの受信値：
|〈---エラー・ループ---〉|
|〈—bzyループ—〉|
ＣＲＣエラー検出
...22 err
使用中検出：
...21 bzy—23 bzy
ノードＢの値：
パケット完了
Bzy Bzy * Bzy Bzy * Done Done
受入れ／拒否：
Rej Rej * Rej Rej * Acpt Acpt
seqTable.seq[producerId]:
20 21 22 * 23 21 22 * 23 24
seqTable.bzseq[producerId]:
0 0 0 * 0 0 0 * 0 1
seqTable.busy[producerId]:
0 1 1 * 1 1 1 * 1 0
seqTable.acpt[producerId]:
20 21 21 * 21 21 21 * 21 24
ノードＣの値：
Packet completion:
(CRC) * Done *
Accept/reject:
--- * Acpt *

45 46

```
                                *                                *
seqTable.seq[producerId]:       *                                *
    20      21      21      *   21      21      22      *   23      24
seqTable.bzseq[producerId]:     *                                *
    0       0       0       *   0       0       0       *   0       1
seqTable.busy[producerId]:      *                                *
                                *                                *
seqTable.acpt[producerId]:      *                                *
    20      21      21      *   21      21      21      *   23      24
```

【０１５１】この例は、パケットがその宛先で受け入れられたかどうかを送信側ノードが検出できないことを示している。この例のノードＣではエラーが発生していないので、パケット２１と２３はどちらも有効なｐＬａｂｅｌ．ｓｅｑフィールドを有し、ＡＣＫ＿ｄｏｎｅ応答＊＊が与えられる。

【０１５２】ケース３：　ノードＢ用の使用中再試行ループを開始する前にノードＣで検出されたノードＣへの送信パケット上のＣＲＣエラー

送信パケットの観点からのパケット順序付け：

47 48

ノードＡの送信値：

宛先：

```
     B   |   C   |   B   |   B   |   C   |   B   |   B   |   B
 pLabel.seq:
     21      22      23  *  21      22      23  *  24      25
                         *                      *
 pLabel.bzseq
     0       0       0   *  0       0       0   *  1       1
                         *                      *
 ＣＲＣエラー再試行：
     21                  *  21                  *  24
     |                      ‘                      ‘
     +---------〉-----+|
 使用中再試行
     +---------〉---------〉-------------+
```

ノードＡの受信値：

```
                           |〈---エラー・ループ---〉|
                                ‘                      |〈—bzyループ—〉|
 ＣＲＣエラー検出
                  ...22 err‘                             ‘
 使用中検出：                                            ‘
                ...21 bzy---23 bzy----------‘
```

```
ノードＢの値：              *                      *
 パケット完了               *                      *
    Bzy             Bzy   *  Bzy             Bzy   *  Done    Done
 受入れ／拒否：
    Rej             Rej   *  Rej             Rej   *  Acpt    Acpt
                          *                      *
 seqTable.seq[producerId]: *
     20      21      22   *  23      21      22   *  23      24
 seqTable.bzseq[producerId]:*                      *
     0       0       0   *  0       0       0   *  0       1
 seqTable.busy[producerId]: *                      *
     0       1       1   *  1       1       1   *  1       0
 seqTable.acpt[producerId]: *                      *
     20      21      21   *  21      21      21   *  21      24
                          *                      *

ノードＣの値：              *                      *
 Packet completion:         *                      *
           (CRC)            *          Done        *
 Accept/reject:
            ---             *          Acpt        *
```

```
                              *                    *
seqTable.seq[producerId]:     *                    *
    20      21      21        *   21      21       22 *   23      24
seqTable.bzseq[producerId]:   *                    *
    0       0       0         *   0       0        0  *   0       1
seqTable.busy[producerId]:    *                    *
                              *                    *
seqTable.acpt[producerId]:    *                    *
    20      21      21        *   21      21       21 *   23      24
```

【０１５３】この例は、以下の一連の事象に基づくものである。ノードＢには２つの送信／応答パケット２１および２３が送信され、そのノードはこれらを使用中として拒否する。ノードＣにはパケット２２が送信されるが、そのノードにＣＲＣエラーがあるので、このパケットはそこで検出されない。ノードＢではエラーが一切検出されない。ノードＡは、まず未解決Ａｃｋの完了を待つことにより、ノードＢのパケットに関してその使用中再試行を開始する。次にノードＡはＣＲＣエラーを検出する。これはＣＲＣエラー・ループを開始するが、このループは２１（エラー前の最後の良好パケット）から２３（エラーが検出される前に最後に送信したパケット）までのすべてのパケットを含まなければならない。

【０１５４】この例では、ＣＲＣエラー再試行ループに入ると、ｐＬａｂｅｌ．ｓｅｑ値およびノードＢのｓｅｑＴａｂｌｅ．ｂｕｓｙ［］フィールドとｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドに基づいて、エラー・ループで再試行したノードＢへのパケットが拒否される。ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］ビットが設定されているので、パケットは拒否される。また、ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］が設定されているが、（ｐＬａｂｅｌ．ｓｅｑ値とｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドとの比較に基づいて）どちらのパケットも最初に「使用中」のマークが付けられているので、再試行したパケット２１と２３はどちらも「使用中」として肯定応答される。

【０１５５】使用中ループが最終的に開始されると、ノードＡからのｓｅｑＴａｂｌｅ．ｂｚｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］フィールドにより変換が行われ、パケット２１がパケット２４として再伝送される。ノードＢはこのトランザクションを検出し、その受信バッファが一杯であるかどうかに基づいて、パケット２４を受け入れられるかどうかを評価する。ここで想定する例では、再試行したパケット２４および２５がどちらも受け入れられる。

【０１５６】潜在的にエラーが存在する場合に使用中再試行ループを開始するための基本概念は、最初に再試行した使用中パケットより前のすべてのパケットの肯定応答が受信され、良好として検査されるまで、ループの開始を待つことである。これが必要である理由は、ｐＬａｂｅｌ．ｂｚｓｅｑビットの状態を変更するとすべてのノード・パケットがそのｓｅｑＴａｂｌｅ．ｂｕｓｙ［］フィールドとｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドをリセットするからである。これらのフィールドは、使用中再試行ループの前にパケット受入れ／拒否および肯定応答を生成するためにエラー再試行状態を維持している。これらのフィールドがリセットされるのは、未解決のままのエラー条件がまったくないことをノードＡが確認した場合に限られる。

【０１５７】作成側ノード：　要求待ち行列状態

好ましい実施形態の各作成側ノードは、１つのレジスタに加えて以下の３通りのタイプの発信側待ち行列と、２通りのタイプの着信側待ち行列または構造（図４０を参照）を有することができる。

発信側：

１．おそらくトランザクション層（リンク層の上）内にあって未発行要求からなる新規要求待ち行列

２．同じくトランザクション層内にあって他のノードへの未発行応答パケットからなる応答保留待ち行列

３．リンク層内にあって、リングレットに対して発行される処理中の送信および応答パケットからなる「ｓｅｎｄ＿ｐｋｔ」待ち行列

４．再試行すべき使用中パケットからなる「ｒｅｔｒｙ＿ｐｋｔ」待ち行列［これは使用中再試行の実施形態にのみ適用されることに留意されたい。］

５．ｒｅｔｒｙ＿ｐｋｔ待ち行列を充填し、エラー再試行用のパケットをｓｅｎｄ＿ｐｋｔ待ち行列にリサイクルするための「ｒｅｔｒｙ＿ｐｋｔ」レジスタ着信側：

１．リンク層内にあってこのノードへの着信肯定応答からなる待ち行列

２．トランザクション層内にあって、応答を待っている要求用の構造

【０１５８】トランザクション層内の発信側送信および応答パケットの場合、場合によっては待ち行列が複数の待ち行列である可能性があり、その待ち行列間に順序付けの依存関係はまったくない。一例としては、スイッチの複数ポートである。

【０１５９】ＳＳＯ順序付けモードでのこのノードへの応答パケットの場合、所与の宛先ノードからの応答は必ず順番に到着するが、異なるノード間の応答に関して順序付けは一切保証されない。その結果、応答を待っている送信パケット構造は、待ち行列ではなくむしろ構造と

して示される。

【０１６０】ＳＳＯ順序付け空間での応答パケットの使用法

ＩＥＥＥ　Ｐ１３９４．２では、宛先ノードにデータを書き込むための動作として、応答なし移動トランザクションと応答付き書込みトランザクションという２通りのタイプの動作を指定している。ＳＳＯ順序付け構造は、順序付けられたデータの配達を保証するものである。移動トランザクションと書込みトランザクションの相違点は何であろうか。

【０１６１】書込み送信パケットは、書込みデータが実際にこのノードに達したことを示す、宛先ノードによって生成された明確な指示を行う。すなわち、これは、動作の完了を明確に示す。移動送信パケットは、完了の明確な指示は行わない。

【０１６２】この相違から２つの質問が発生する。

１．完了の明確な指示は何のために使用できるか。

２．特に無効アドレスと、新しいパケットの受入れを拒否する「膠着」受信側ノードとを含む、不完全なリンク・トポロジで順序付けが維持されていることを確認するために、どのようなアーキテクチャ上の拡張を移動トランザクションと結合することができるか。書込みトランザクションに関する完了の明確な指示は、トランザクション層から見えるものになる。これより上の何らかのシステム・レベルでこれが見えるようになるかどうかは、この説明の範囲を超えたトランザクション・レベルのアーキテクチャによって決まる。ＳＳＯ順序付けのポイントは、順序付けを確保するためにいかなる構造も不要であるということである。

【０１６３】トランザクション層内の書込みトランザクションに可能な使い方は２通りある。このリストはほとんど網羅的ではない。第１に、書込みトランザクションは信頼性アーキテクチャ用の基礎として使用することができる。送信パケットが指定の期間内に応答を受信できなかった場合、ノード・タイムアウトが報告されるはずである。この使用法では、次の送信パケットを開始する前に応答を完了するための事前要件はまったく存在しない。

【０１６４】書込みトランザクション用の第２の使用法は、可用性の高いアーキテクチャ用の基礎として使用することである。この応用例では、書込みに対する応答が「書込みコミット」の完了を安定した記憶装置に通知する。これに関する問題は、どのように応答を処理するかということである。それを使用して次の送信パケットの伝送をゲートするのだろうか。そのような場合、これにより、パケット配達速度は大幅に低下する。システム・アーキテクチャは、完了バリア・インジケータとして応答を使用するためのメカニズムをサポートしているか。ほとんどは（ＵＰＡを含む）サポートしていない。

【０１６５】送信パケット待ち行列構造

Ｐ１３９４．２に応じたアドレス空間は、一般に、順序付けされる場合もあれば、順序付けされない場合もあるので、ノードは、両方のアドレス指定モデルを識別しサポートするために各コマンドごとにデータの構造を維持しなければならない。この構造案は以下のように３つのフィールドからなる。

Send_pktＳＳＯ構造、送信または応答パケット当たり１つずつ

```
send_pkt.val        (1)  /* 項目有効 */
send_pkt.sso        (1)  /* コマンド・アドレス空間から */
send_pkt.done       (1)  /* ACK_doneを受信、応答が必要な場合もある*/
send_pkt.busy       (1)  /* ACK_busyを受信、再試行待ち */
send_pkt.init_er    (1)  /* エラー再試行ループ内の初期（第１の）パケット */
send_pkt.no_xmit    (1)  /* エラー・ループ中にリングレットへの伝送なし*/
send_pkt.err_ret    (1)  /* エラー再試行ループ内のエラー再試行状態*/
send_pkt.seq        (6)  /* この送信／応答パケット用の順序番号 */
send_pkt.packet     (N)  /* Ｎビットの送信／受信パケット */
```

その他の制御ビットは、ここに記載するＳＳＯ順序付けを上回る実施目的に使用することができる。

【０１６６】概念上、ｓｅｎｄ＿ｐｋｔ[]はＦＩＦＯ待ち行列と見なすことができ、対応するＦＩＦＯ内の肯定応答パケットが受信されると、新たに送信されたパケットは一番上に入り、終了したパケットは一番下から出ていく。「ａｃｋ＿ｄｏｎｅ」として肯定応答された送信パケットは、応答を待っている可能性がある。

【０１６７】ｓｅｎｄ＿ｐｋｔＦＩＦＯ待ち行列に加え、他の待ち行列、レジスタ、ＭＵＸ、状態マシン、制御論理については、補助の流れ図に示す。

【０１６８】ＣＲＣエラー・ループ用の作成側ノード・ポインタ

ＣＲＣエラー・ループおよび使用中再試行ループの処理には、所与の作成側ノード変数が必要である。ＣＲＣエラー・ループ動作用のグローバル変数として以下のものがある。この場合、接頭部「ｏｒ＿」は「順序付き」を意味する。

```
or_last_ct          (6)  /* ローカル・リングレットに送信された最新ＳＳＯ順序付きｐｋｔ用の順序カウント */
no_last_ct          (6)  /* ローカル・リングレットに送信された最新非ＳＳＯ順序付きｐｋｔ用の順序カウント */
or_rcvd_ct          (6)  /* ローカル・リングレットか
```

ら受信した最新ＳＳＯ順序付きｐｋｔ用の順序カウント */

【０１６９】ローカル・リングレットにパケットが送信されると、パケット・パスでｓｅｎｄ＿ｐｋｔレジスタ内に伝達されるｓｅｎｄ＿ｐｋｔフィールドと一致するように、そのｐＬａｂｅｌフィールドが設定される。一般に、以下のようになる。

```
pLabel.sso     = send_pkt_reg.sso
pLabel.bzseq   = bzseq /* グローバル作成側ノードの状態ビット */
pLabel.seq     = send_pkt_reg.seq
```

ｐＬａｂｅｌ．ｓｅｑにどの値が割り当てられるかは、動作（たとえば、新しいパケット、使用中再試行パケット、またはＣＲＣ再試行）によって決まる。以下に示す詳細を参照されたい。

【０１７０】受信側ポインタ比較：折返しモジュロ６４＊

```
if {pLabel.seq <= seqTable.seq[producerId]} then
   {seqTable.seq[producerId] = pLabel.seq
   }
```

３．最後に、各ノードは、以下の場合に、ｐｒｏｄｕｃｅｒＩｄから順序エラーを含むパケットを検出することができる。

```
pLabel.seq > seqTable.seq[producerId] + 1
```

【０１７１】モジュロ６４カウンタによって「より大」比較と「より小か等しい」比較の両方をサポートすることは、この比較が３１という受信側ノードにおける差を超えないことをｐｒｏｄｕｃｅｒＩｄによって確認した場合に達成することができ、カウントは以下のように定義される。

以下の場合は「Ａ＜＝Ｂ」
－上位ビットが等しく、下位ビットがＡ＜＝Ｂ
－上位ビットが等しくなく、残りのビットがＡ＞Ｂ
以下の場合は「Ａ＞Ｂ」
－上位ビットが等しく、下位ビットがＡ＞Ｂ
－上位ビットが等しくなく、残りのビットがＡ＜＝Ｂ

【０１７２】特殊な問題

ここでは、本発明の可能な実施態様を含むある程度までの詳細に関する追加の説明と重要性を示す。

【０１７３】このような問題のうちの第１の問題では、ＣＲＣエラー後に適切なｓｅｑＴａｂｌｅ［ｐｒｏｄｕｃｅｒＩｄ］フィールドを確実にリセットすることを扱っていた。ＣＲＣエラーが発生すると、リングレット内のすべてのノード用のｓｅｑＴａｂｌｅ．ｓｅｑ［ｐｒｏｄｕｃｅｒＩＤ］内のフィールドは、リングレット内の他のノードと同期していない値を含む可能性がある。

【０１７４】同時に、ｓｅｑＴａｂｌｅ［］の値は、有効かつ正しく順序付けられたパケットが受信側ノードで識別された場合「のみ」更新されるので、これらの値はこのノードに関する限り正しいと見なさなければならない。しかし、これらは、送信側ノードと他のリングレット・ノードの両方に対して同期していない場合もある。

【０１７５】したがって、ｐｒｏｄｕｃｅｒＩｄノードの仕事は、すべてのノードを確実に同期状態に戻すことである。問題はこのリセットを達成する際に発生する。すなわち、エラー・ループ自体の第１のパケットにＣＲＣエラーが発生したらどうだろうか。

【０１７６】この問題に対する単純な解決策は、非パイプライン化モードで最後の既知の良好完了パケットを伝送することにより、エラー処理ループを実行するｐｒｏｄｕｃｅｒＩｄノードがそのエラー・ループを開始することである。ｐｒｏｄｕｃｅｒＩＤノードは、それが有効な肯定応答を受信するまで、このパケットを再試行するはずである。これを受信すると、リング上のすべてのノードは、このｐｒｏｄｕｃｅｒＩＤ用のｓｅｑＴａｂｌｅ．ｓｅｑ［］フィールドが同じ値に設定されていなければならない。ｓｅｑＴａｂｌｅ．ｂｚｓｅｑ［］フィールドをリセットするために同じ引数が適用される。しかし、ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］フィールドは、このｐｒｏｄｕｃｅｒＩｄからこの受信側ノードに配達された最後の有効完了パケットへの肯定応答の状態を反映する。次の有効パケットの処理方法を考慮する場合（再試行したパケットは拒否される）、このフィールドは有効性を有する。その結果、このフィールドは、リセット間隔中にリセットしてはならない。同様に、ｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドも未変更のままになる。

【０１７７】ただし、既知の良好完了パケット（有効として肯定応答され、リングレット内のターゲット・ノードによって拒否されるはずのもの）を再試行することにより、ｐｒｏｄｕｃｅｒＩＤノードは、そのデータ内容ではなく、そのパケット順序番号のみに関するパケット

＊カウントのサポート

上記の「提案の概要」の項により、受信側の動作にとって３通りの比較が重要になる可能性があることに留意されたい。

１．受信側ノードは、以下の場合に、所与のｐｒｏｄｕｃｅｒＩｄから有効な新しいＳＳＯ順序付きパケットを検出することができる。

```
pLabel.seq = seqTable.seq[producerId] + 1　かつ
pLabel.seq > seqTable.acpt[producerId]
```

２．このｐｒｏｄｕｃｅｒＩｄがＣＲＣエラー・ループを実行していることをこのノードが検出すると、ｓｅｑＴａｂｌｅ．ｓｅｑ［ｐｒｏｄｕｃｅｒＩｄ］フィールドは現行のｐＬａｂｅｌ．ｓｅｑ値にリセットされる。これは、このノードからのパケットまたは肯定応答用のｓｅｑフィールドを比較することによって示される。

を伝送することに留意されたい。

【０１７８】有効なＡＣＫが返されるまで既知のパケットを再送信するという同じ手法を使用すると、リングレット初期設定によって上記のフィールドを初期設定することができる。しかし、ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］およびｓｅｑＴａｂｌｅ．ａｃｐｔ［］などの他のフィールドは、非ＳＳＯ空間を使用してｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドに値（ｐＬａｂｅｌ．ｓｅｑ）を書き込むなど、他の手段によってリセットしなければならない。

【０１７９】元の提案に関する第２の問題では、そのエラー・ループ内のパケットが完了したかどうかを判定するｐｒｏｄｕｃｅｒＩＤノードの曖昧さの問題を扱っていた。すなわち、（送信／受信）と肯定応答の対にＣＲＣエラーが発生していると検出された場合、ｐｕｒｏｄｕｃｅｒＩＤノードは、パケットがそのリングレット宛先に達する前または後のいずれでＣＲＣエラーが発生したかを判定することができない。したがって、これは、再試行したパケットに「完了」というマークを確実に付けることはできない。

【０１８０】この問題に対する解決策は、（６ビットの）ｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドを各受信側ノードのｓｅｑＴａｂｌｅアレイに追加し、このｐｒｏｄｕｃｅｒＩｄからのより小さい値のｐＬａｂｅｌ．ｓｅｑを含むパケットによってそのｓｅｑＴａｂｌｅ．ｓｅｑ［］がリセットされる前にこのノードで最後に検出された有効順序番号を記録し、または使用中再試行のために拒否された第１のパケットの順序番号（ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］フィールドによって示される）を記録することである。ただし、ｓｅｑＴａｂｌｅ．ａｃｐｔ［］フィールドを追跡することは、以下の意味を有することに留意されたい。

１．このノードで最後に受け入れられたパケットはｓｅｑＴａｂｌｅ．ａｃｐｔ［］と等しいかそれより小さい順序番号を備えていなければならず、したがって、新しい未検出のパケットはｓｅｑＴａｂｌｅ．ａｃｐｔ［］より大きい順序番号を備えていなければならない。ただし、ｓｅｑＴａｂｌｅ．ｂｕｓｙ［］はリセットされるものとする。

２．エラーまたは使用中パケットがない場合、ｓｅｑＴａｂｌｅ．ａｃｐｔは現行のｐＬａｂｅｌ．ｓｅｑ値を追跡する。この事実は、モジュロ６４のローリング・カウント定義を作成する際に非常に重要なものである。

【０１８１】図４０～６０の例に関する注意事項

図４０～６０は、パケット処理の連続段階における本発明のシステムの状態を示しており、その場合、使用中条件は消費側ノードに関して発生し、ＣＲＣ（またはその他の）エラー条件も発生し、「使用中」ノードは挙動が不適切である。すなわち、何らかの所定の長さの時間または所定の数の送信または受信パケットの間、それが持続的に使用中であったか、またはそれが間違ってａｃｋ＿ｂｕｓｙ応答を伝送している。

【０１８２】図４０～６０の例では、ｏｒ＿ｌａｓｔ＿ｃｔは最後に送信した順序付きパケットのカウント数であり、ｏｒ＿ｒｃｖｄ＿ｃｔは最後に受信した順序付きパケットのカウントを指すことに留意されたい。使用中再試行が必要な場合（再送信論理で検出される）、ｏｒ＿ｒｃｖｄ＿ｃｔ値はフリーズされる。

【０１８３】この例では、以前の使用中パケット、たとえば、５４と５９がｒｅｔｒｙ＿ｐｋｔ待ち行列内に入っていると想定する。ただし、以下の式が成り立つことに留意されたい。

$$(or\_last\_ct - or\_rcvd\_ct) = (23-59) \bmod 64 = 28$$

また、２８はその差（ｏｒ＿ｌａｓｔ＿ｃｔ － ｏｒ＿ｒｃｖｄ＿ｃｔ）のしきい値と等しいかそれより小さい。

【図面の簡単な説明】

【図１】より大規模なネットワーク上のリングレットを示すブロック図である。

【図２】単一リングレットを示すブロック図であって、そのリングレット上で送受信するコマンドを示す図である。

【図３】強力順次順序付け（ＳＳＯ）の場合に送受信するコマンドを示す図である。

【図４】非アイデンポテント要求またはコマンドを伴うＳＳＯの場合に送受信するコマンドを示す図である。

【図５】図２のリングレットを示し、送信パケットおよび肯定応答のデータ構造を示す図である。

【図６】図２のリングレットを示し、パケットの再送信を示す図である。

【図７】図２のリングレットを示し、消費側ノードのアクションを示す図である。

【図８】送信インタフェースの観点から本発明の動作を示すブロック図である。

【図９】送信ポートに関して本発明の使用中ループ動作を示すタイミング図である。

【図１０】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１１】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１２】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１３】本発明のシステムの論理サブシステムとその

動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１４】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１５】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１６】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１７】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１８】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図１９】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２０】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２１】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２２】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２３】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２４】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２５】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２６】本発明のシステムの論理サブシステムとその動作を示す論理図であって、（非公式図面では）六角形は判断ボックスであり、矩形は可変値が設定されるステップであることを示す図である。

【図２７】受信インタフェースの観点から本発明の動作を示すブロック図である。

【図２８】受信パケット・タイミングを示すタイミング図である。

【図２９】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３０】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３１】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３２】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３３】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３４】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３５】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３６】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３７】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３８】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図３９】受信ノードに関して本発明のシステムの論理サブシステムとその動作を示す論理図である。

【図４０】本発明の好ましい実施形態におけるパケットの構造と流れの両方を示すブロック図である。

【図４１】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４２】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４３】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４４】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４５】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４６】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４７】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４８】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図４９】本発明のシステムの動作による図４０のブロ

59

ック図の後続状態を示す図である。

【図５０】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５１】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５２】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５３】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５４】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５５】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５６】本発明のシステムの動作による図４０のブロ＊

60

＊ック図の後続状態を示す図である。

【図５７】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５８】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図５９】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【図６０】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

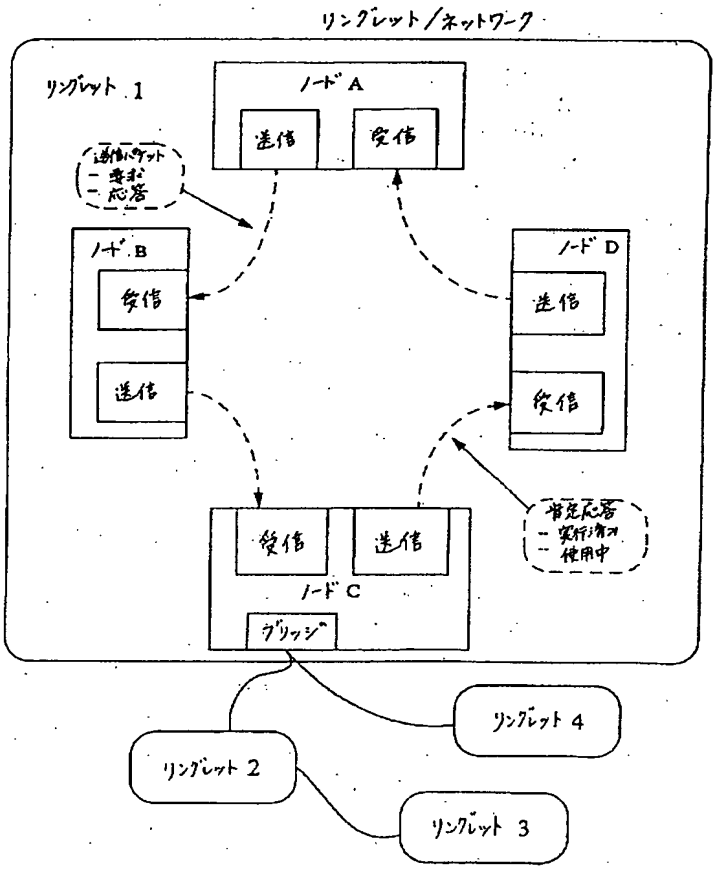
【図６１】本発明のシステムの動作による図４０のブロック図の後続状態を示す図である。

【符号の説明】

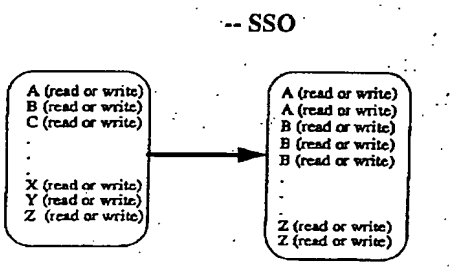
１～４　リングレット１

Ａ～Ｄ　ノード

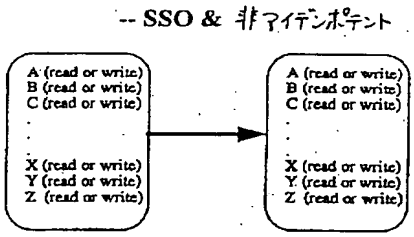
【図１】



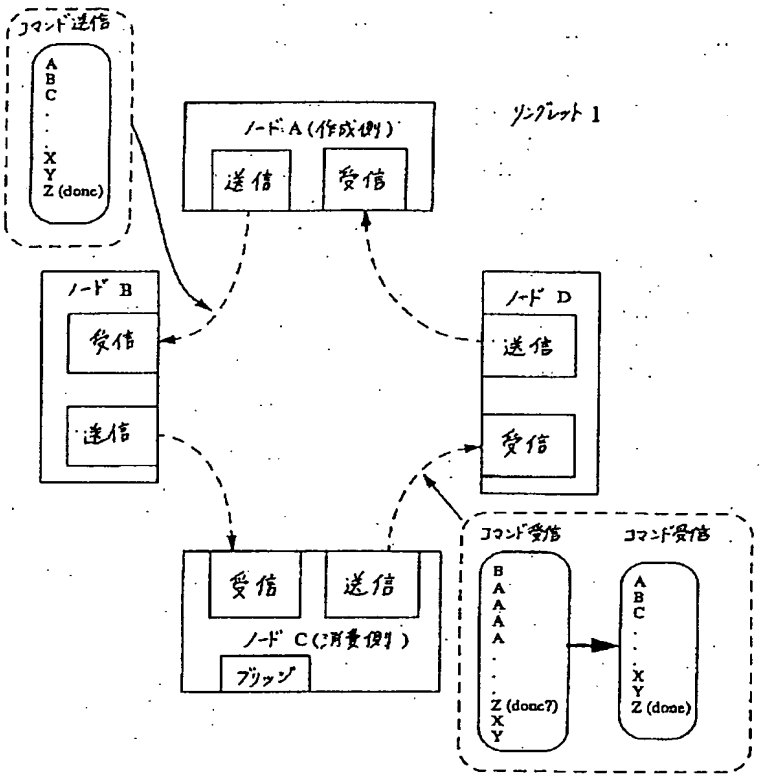
【図３】



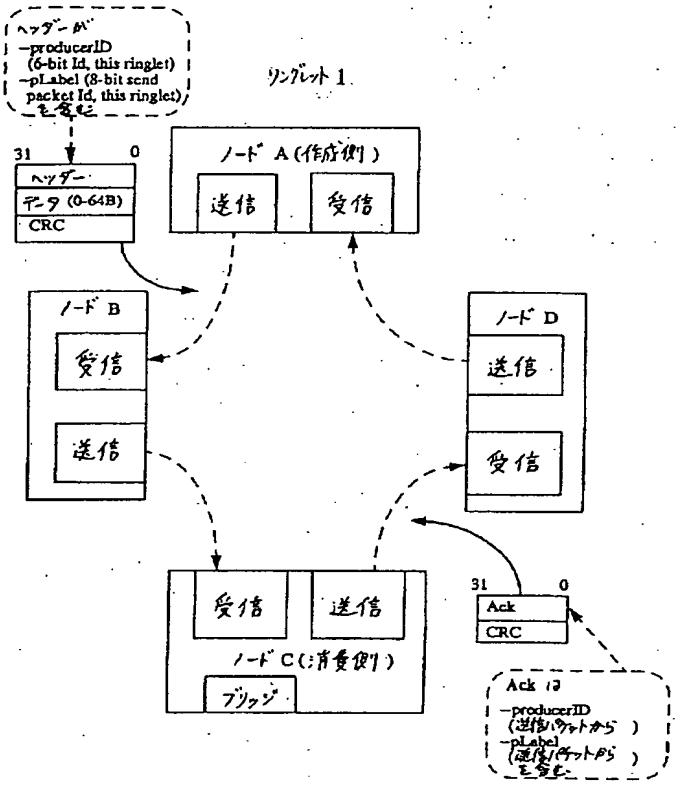
【図４】



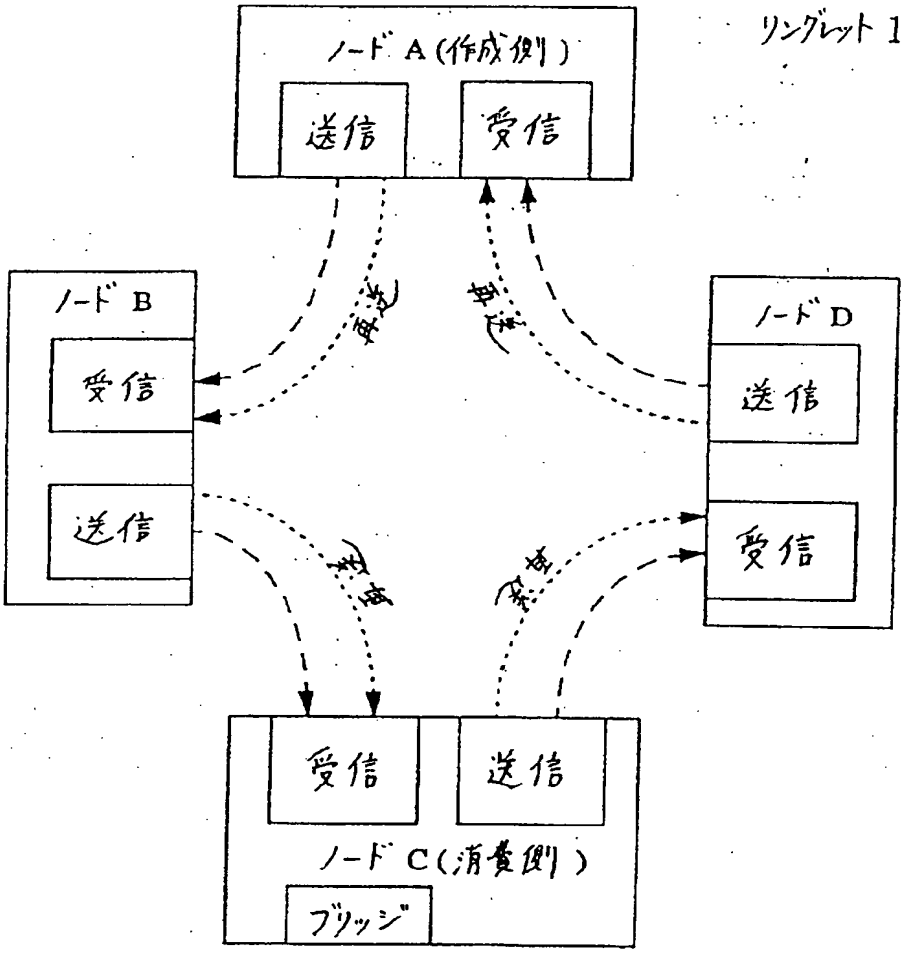
【図２】


コマンド送信
A
B
C
・
・
・
X
Y
Z (done)
ノード A (作成側)
送信
受信
ソングレット 1
ノード B
受信
送信
ノード D
送信
受信
受信
送信
ノード C (消費側)
ブリッジ
コマンド受信
B
A
A
A
A
・
・
・
Z (done?)
X
Y
コマンド受信
A
B
C
・
・
・
X
Y
Z (done)

【図５】


ヘッダーが
–producerID
(6-bit Id, this ringlet)
–pLabel (8-bit send
packet Id, this ringlet)
を含む
リングレット１
31
0
ヘッダー
データ (0-64B)
CRC
ノード A (作成側)
送信
受信
ノード B
受信
送信
ノード D
送信
受信
受信
送信
ノード C (消費側)
ブリッジ
31
0
Ack
CRC
Ack は
–producerID
(送信パケットから)
–pLabel
(送信パケットから
を含む

【図６】


ノード A (作成側)
送信
受信
リングレット 1
ノード B
受信
送信
ノード D
送信
受信
受信
送信
ノード C (消費側)
ブリッジ

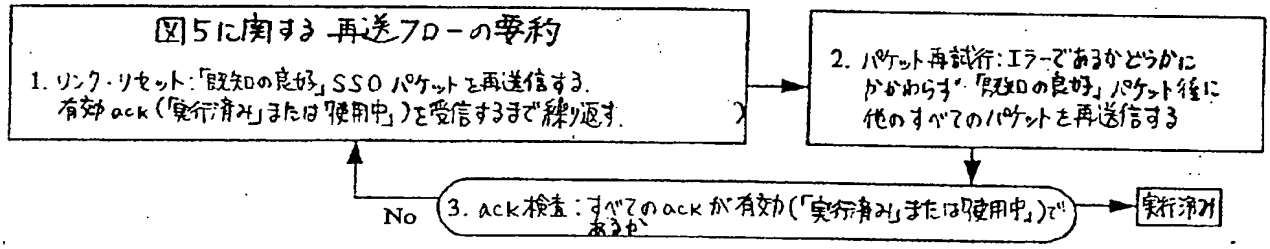
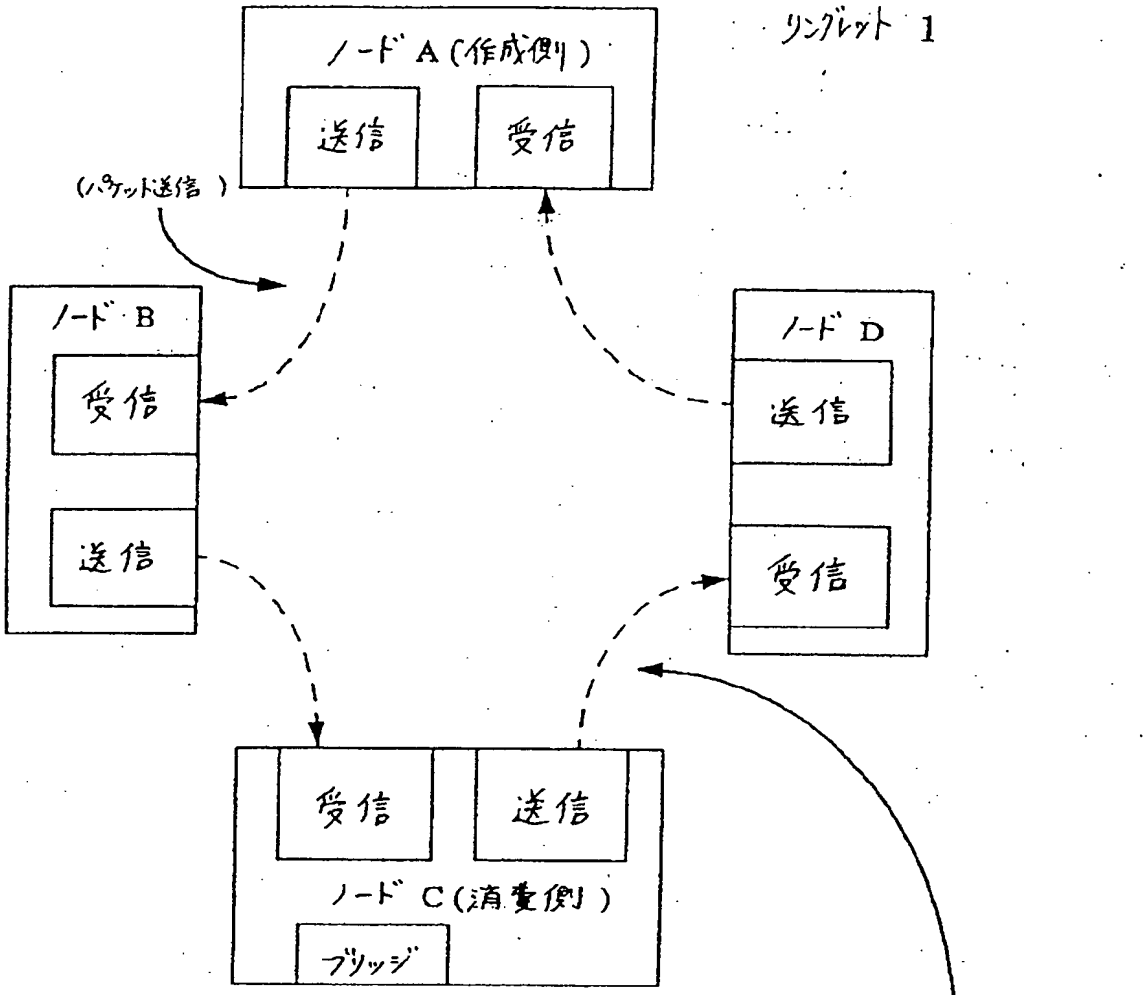


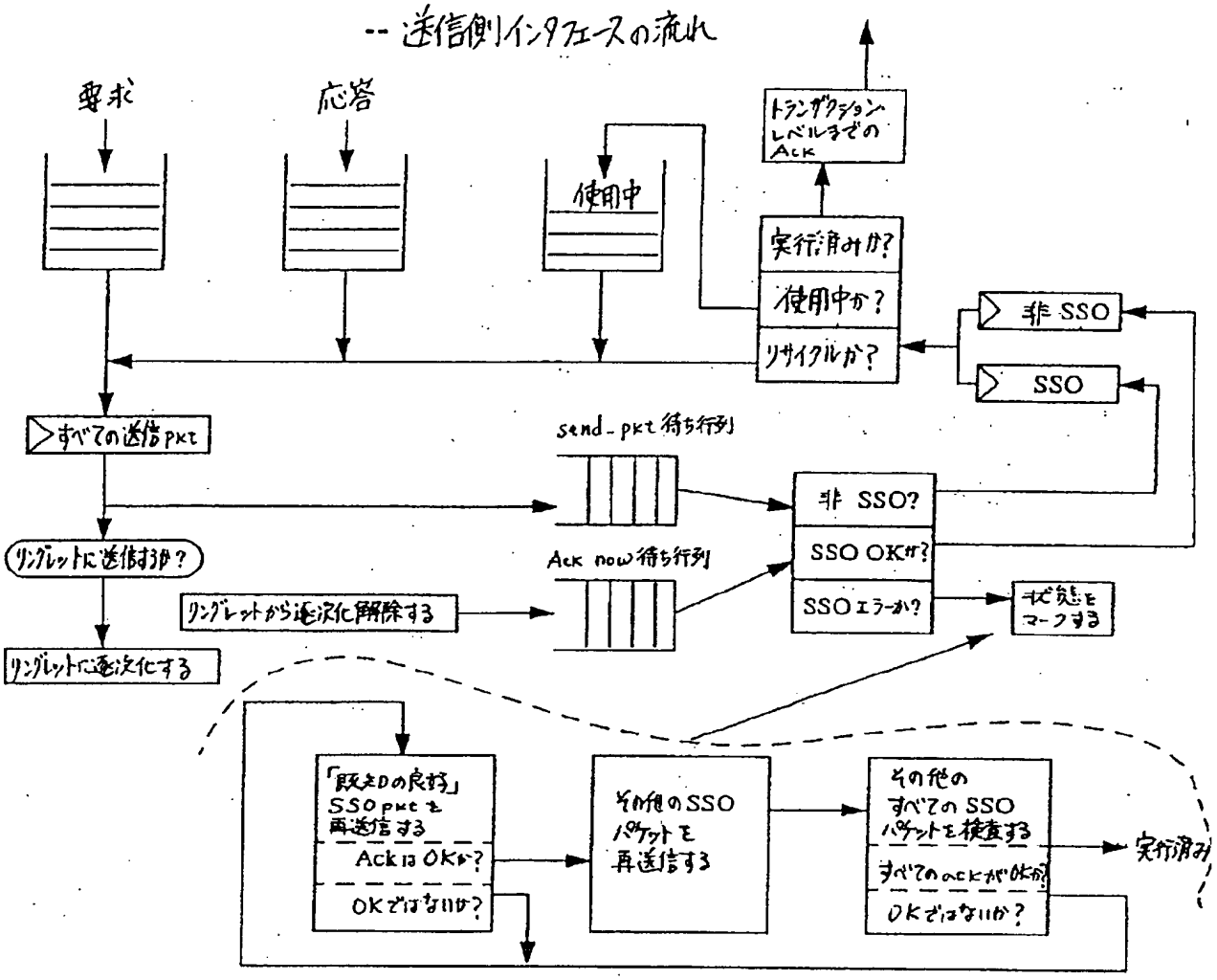
図5に関する再送フローの要約
1. リンク・リセット：「既知の良好」SSO パケットを再送信する．
有効 ack（「実行済み」または「使用中」）を受信するまで繰り返す．
2. パケット再試行：エラーであるかどうかにかかわらず「既知の良好」パケット後に他のすべてのパケットを再送信する
No
3. ack検査：すべての ack が有効（「実行済み」または「使用中」）であるか
実行済み

【図７】



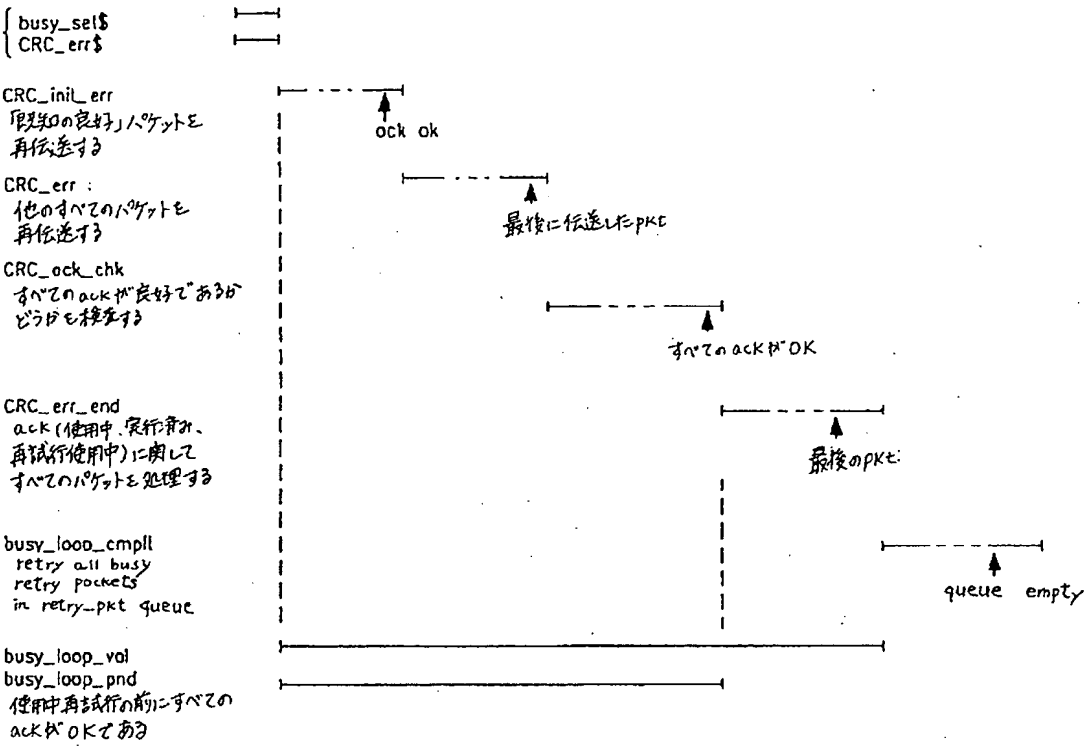
消費側アクション	新しいパケットの肯定応答	再送信パケットの肯定応答
パケットを受け入れる	Ack 実行済み	—
パケットを拒否する	Ack 使用中	Ack実行済み Ack使用中

【図８】


-- 送信側インタフェースの流れ
要求
応答
使用中
トランザクション・レベルまでのAck
実行済みか?
使用中か?
リサイクルか?
非 SSO
SSO
すべての送信pkt
send_pkt待ち行列
非 SSO?
SSO OKか?
SSOエラーか?
リングレットに送信するか?
Ack now待ち行列
リングレットから送信化解除する
リングレットに送信化する
「既知の良好」SSO pktを再送信する
Ackは OKか?
OKではないか?
その他のSSOパケットを再送信する
その他のすべてのSSOパケットを検査する
すべてのACKがOKか?
OKではないか?
実行済み

【図９】

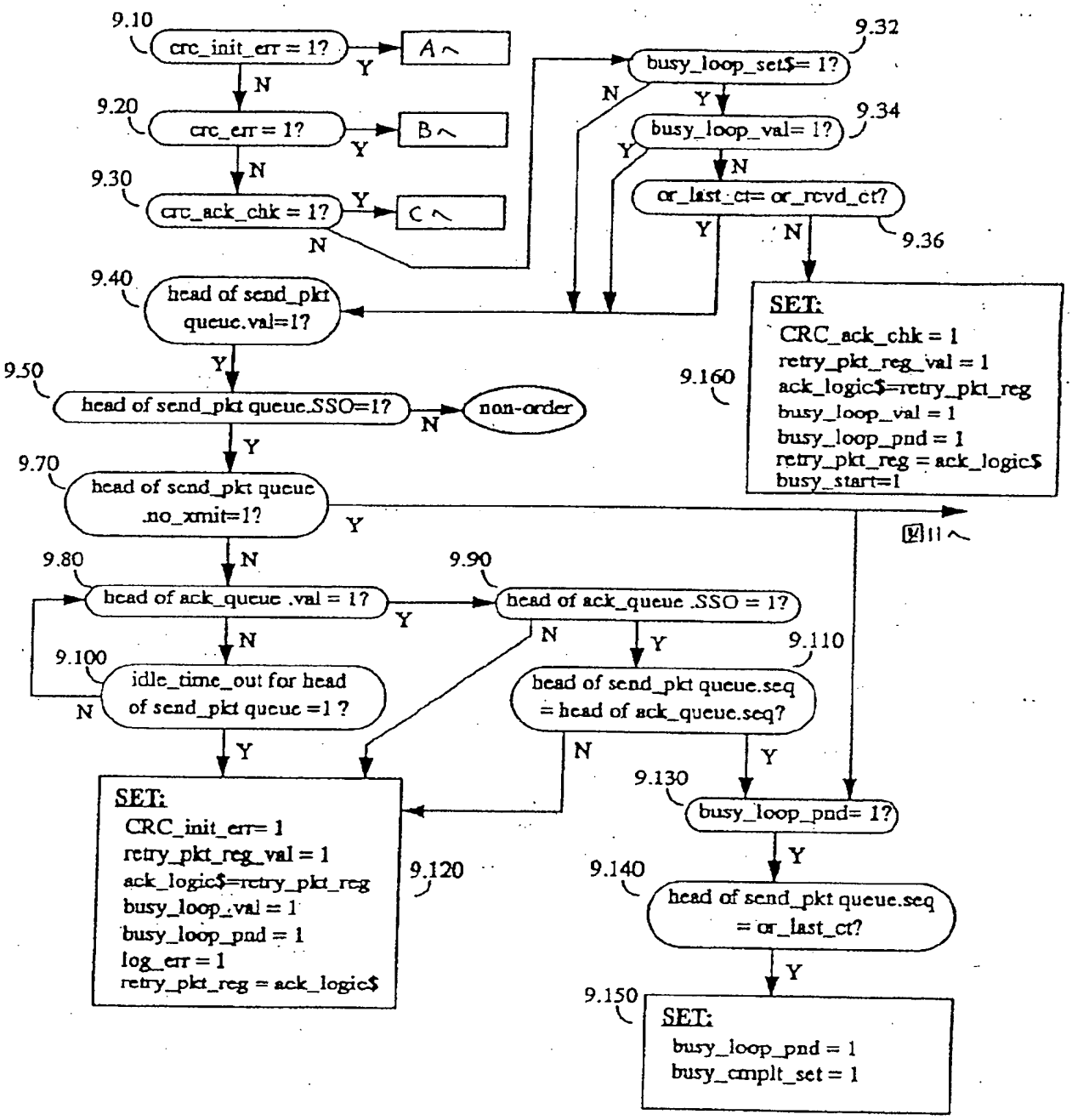
使用中ループ・タイミング図－－送信ポート



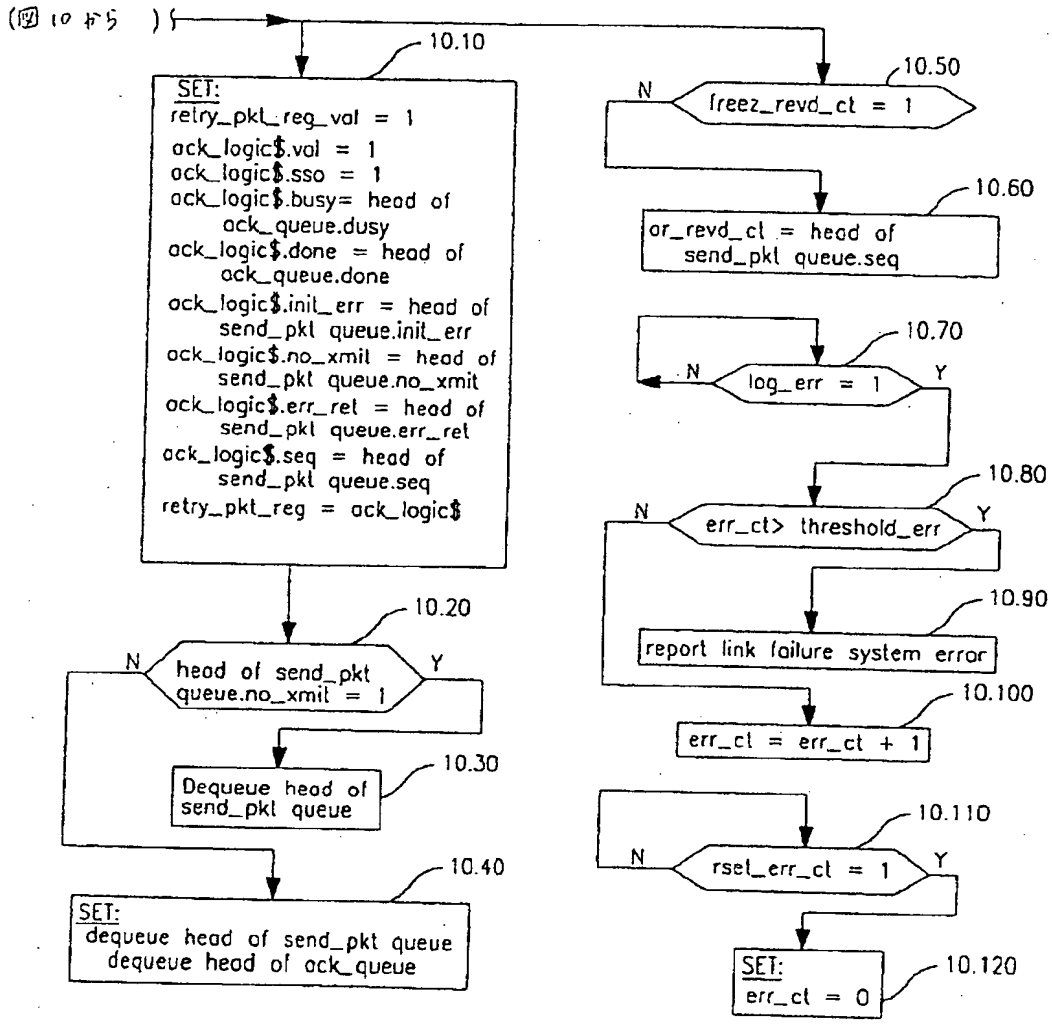
エラー・ループ・パケット状態割当て

	CRC_init_err	CRC_err	CRC_ack_chk
第1のパケット 「既知の良好」	init_err	init_err no_xmit	init_err no_xmit
中間パケット	no_xmit	err_rel	no_xmit
最後のパケット	no_xmit	err_rel	err_rel no_xmit

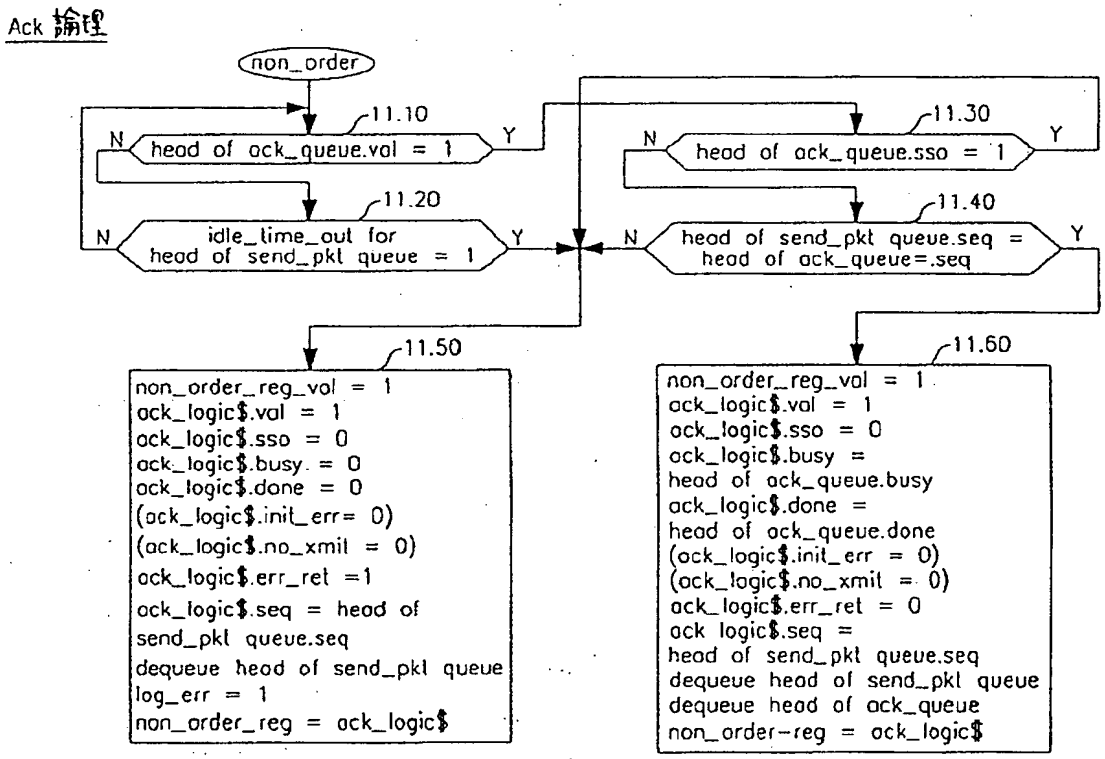
【図１０】


9.10
crc_init_err = 1?
Y
A へ
N
9.20
crc_err = 1?
Y
B へ
N
9.30
crc_ack_chk = 1?
Y
C へ
N
9.32
busy_loop_set$= 1?
N
Y
9.34
busy_loop_val= 1?
Y
N
or_last_ct= or_rcvd_ct?
Y
N
9.36
9.40
head of send_pkt queue.val=1?
Y
9.50
head of send_pkt queue.SSO=1?
N
non-order
Y
9.70
head of send_pkt queue .no_xmit=1?
Y
9.160
SET:
CRC_ack_chk = 1
retry_pkt_reg_val = 1
ack_logic$=retry_pkt_reg
busy_loop_val = 1
busy_loop_pnd = 1
retry_pkt_reg = ack_logic$
busy_start=1
図11へ
N
9.80
head of ack_queue .val = 1?
Y
N
9.90
head of ack_queue .SSO = 1?
N
Y
9.100
idle_time_out for head of send_pkt queue =1 ?
N
Y
9.110
head of send_pkt queue.seq = head of ack_queue.seq?
N
Y
SET:
CRC_init_err= 1
retry_pkt_reg_val = 1
ack_logic$=retry_pkt_reg
busy_loop_val = 1
busy_loop_pnd = 1
log_err = 1
retry_pkt_reg = ack_logic$
9.120
9.130
busy_loop_pnd= 1?
Y
9.140
head of send_pkt queue.seq = or_last_ct?
Y
9.150
SET:
busy_loop_pnd = 1
busy_cmplt_set = 1

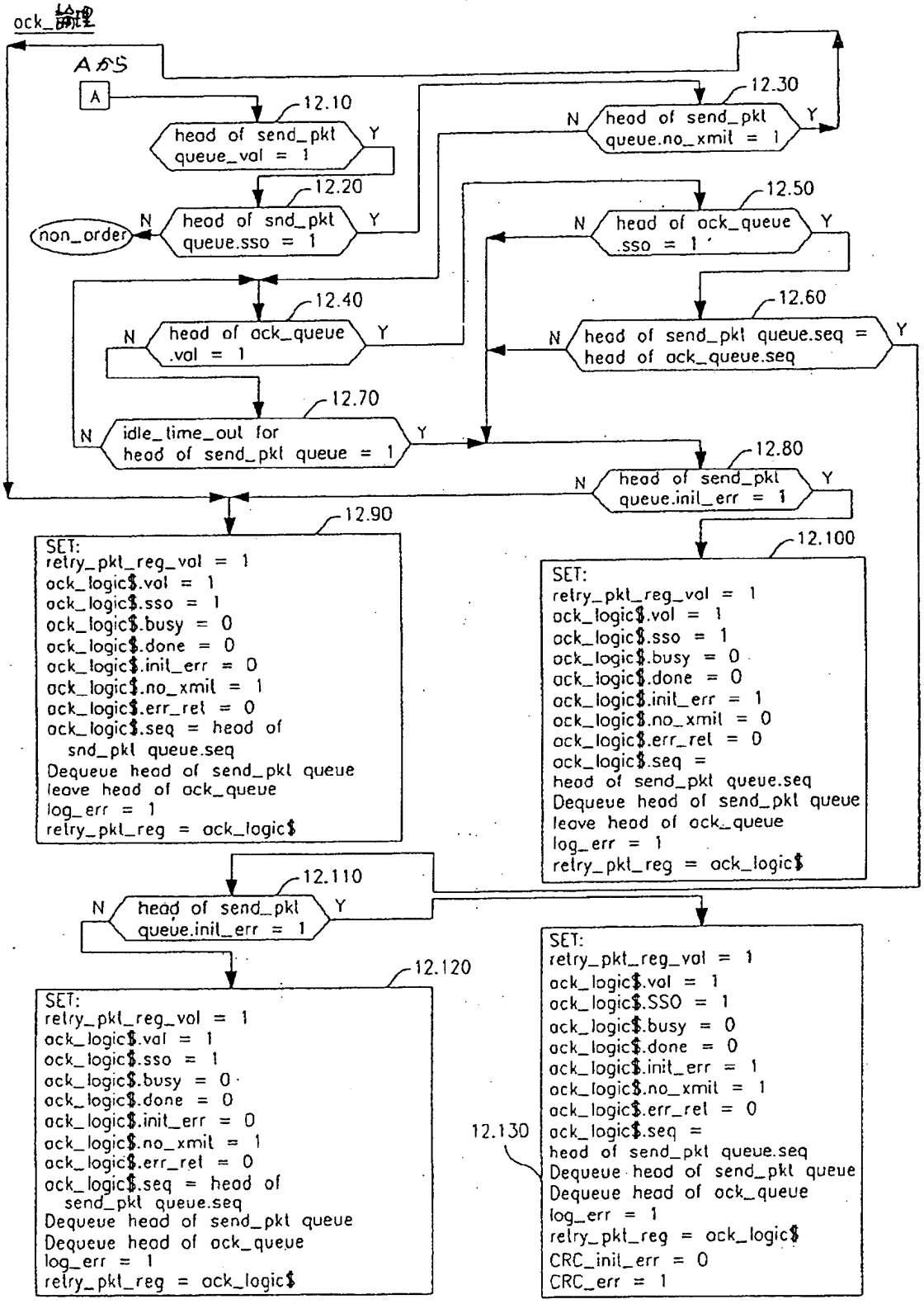
【図１１】


(図10から)
10.10
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy= head of ack_queue.dusy
ack_logic$.done = head of ack_queue.done
ack_logic$.init_err = head of send_pkt queue.init_err
ack_logic$.no_xmit = head of send_pkt queue.no_xmit
ack_logic$.err_ret = head of send_pkt queue.err_ret
ack_logic$.seq = head of send_pkt queue.seq
retry_pkt_reg = ack_logic$
10.20
N
head of send_pkt queue.no_xmit = 1
Y
Dequeue head of send_pkt queue
10.30
10.40
SET:
dequeue head of send_pkt queue
dequeue head of ack_queue
10.50
N
freez_revd_ct = 1
10.60
or_revd_ct = head of send_pkt queue.seq
10.70
N
log_err = 1
Y
10.80
N
err_ct> threshold_err
Y
10.90
report link failure system error
10.100
err_ct = err_ct + 1
10.110
N
rset_err_ct = 1
Y
SET:
err_ct = 0
10.120

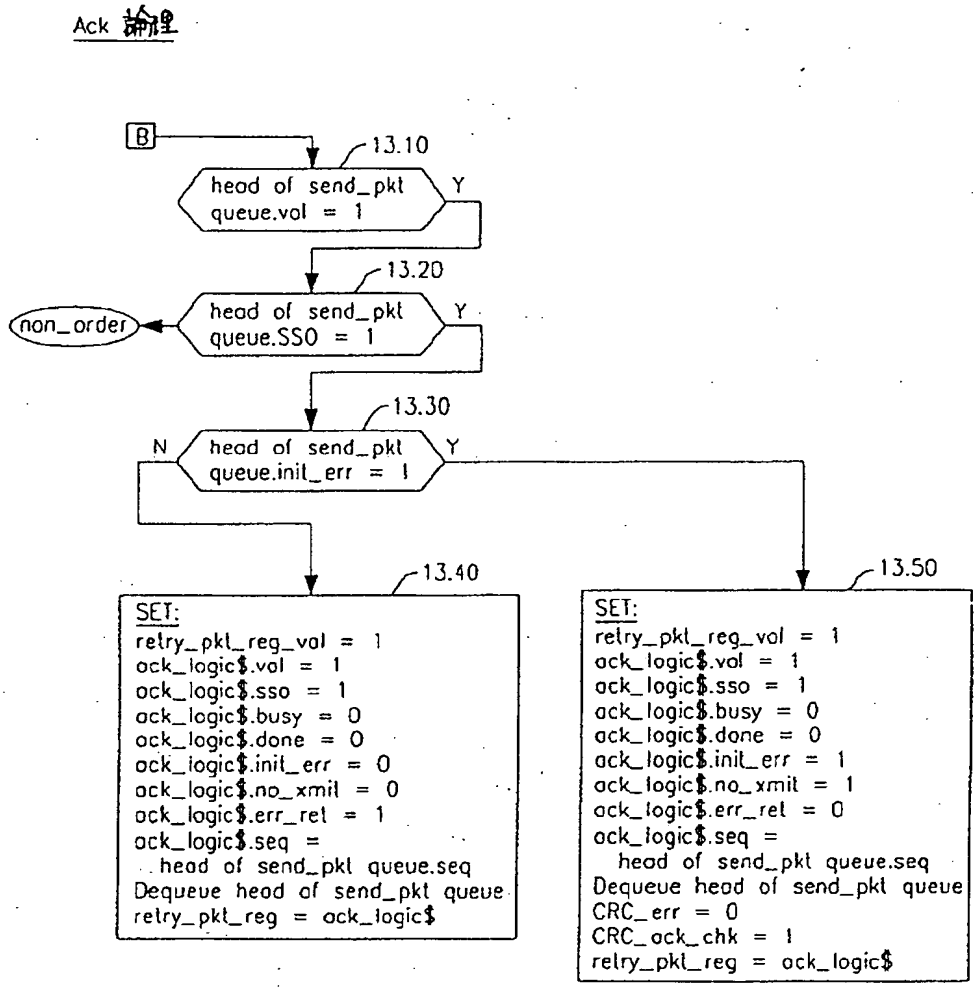
【図１２】


Ack 論理
non_order
11.10
head of ack_queue.val = 1
N
Y
11.20
idle_time_out for
head of send_pkt queue = 1
N
Y
11.30
head of ack_queue.sso = 1
N
Y
11.40
head of send_pkt queue.seq =
head of ack_queue=.seq
N
Y
11.50
non_order_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 0
ack_logic$.busy = 0
ack_logic$.done = 0
(ack_logic$.init_err= 0)
(ack_logic$.no_xmit = 0)
ack_logic$.err_ret =1
ack_logic$.seq = head of
send_pkt queue.seq
dequeue head of send_pkt queue
log_err = 1
non_order_reg = ack_logic$
11.60
non_order_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 0
ack_logic$.busy =
head of ack_queue.busy
ack_logic$.done =
head of ack_queue.done
(ack_logic$.init_err = 0)
(ack_logic$.no_xmit = 0)
ack_logic$.err_ret = 0
ack logic$.seq =
head of send_pkt queue.seq
dequeue head of send_pkt queue
dequeue head of ack_queue
non_order-reg = ack_logic$

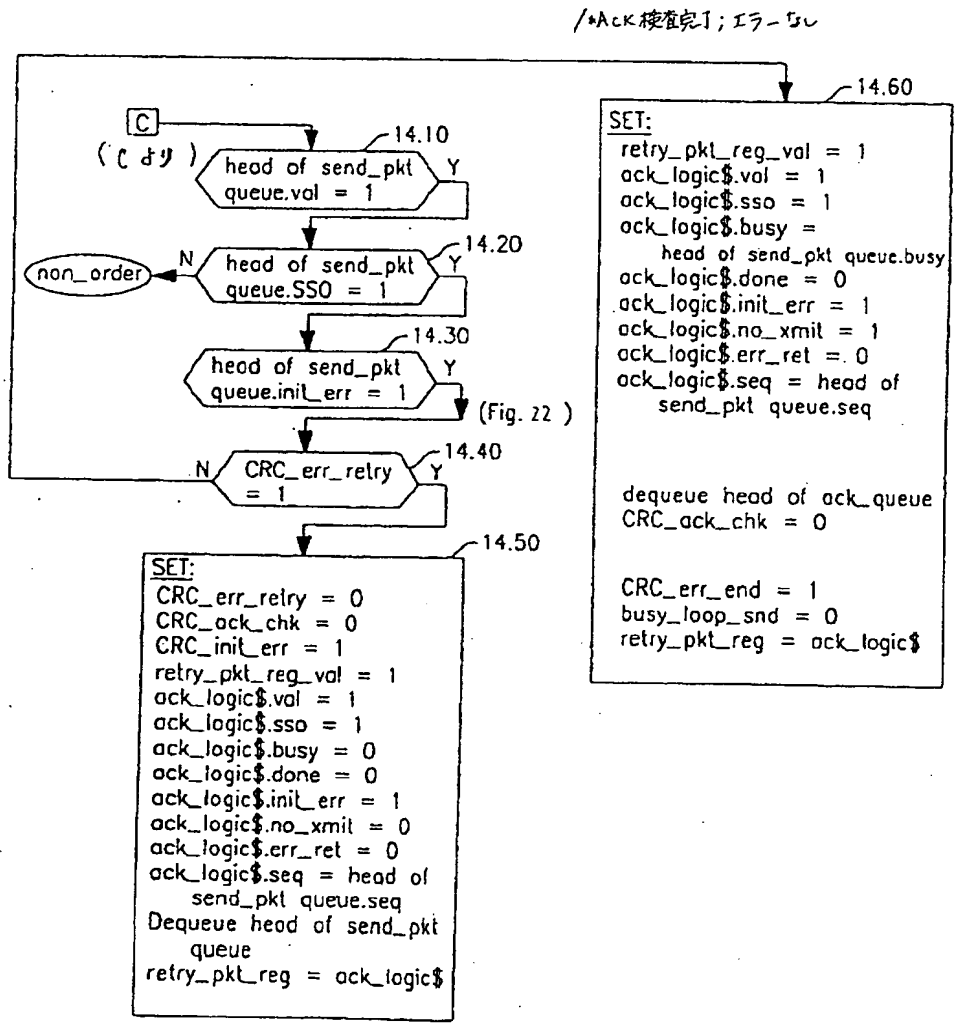
【図１３】


ack_処理
AからS
A
12.10
head of send_pkt queue_val = 1
Y
12.20
non_order
N
head of snd_pkt queue.sso = 1
Y
12.30
N
head of send_pkt queue.no_xmit = 1
Y
12.40
N
head of ack_queue .val = 1
Y
12.50
N
head of ack_queue .sso = 1
Y
12.60
N
head of send_pkt queue.seq = head of ack_queue.seq
Y
12.70
N
idle_time_out for head of send_pkt queue = 1
Y
12.80
N
head of send_pkt queue.init_err = 1
Y
12.90
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 0
ack_logic$.no_xmit = 1
ack_logic$.err_rel = 0
ack_logic$.seq = head of snd_pkt queue.seq
Dequeue head of send_pkt queue
leave head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
12.100
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 0
ack_logic$.err_rel = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
leave head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
12.110
N
head of send_pkt queue.init_err = 1
Y
12.120
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 0
ack_logic$.no_xmit = 1
ack_logic$.err_rel = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
Dequeue head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
12.130
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.SSO = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 1
ack_logic$.err_rel = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
Dequeue head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
CRC_init_err = 0
CRC_err = 1

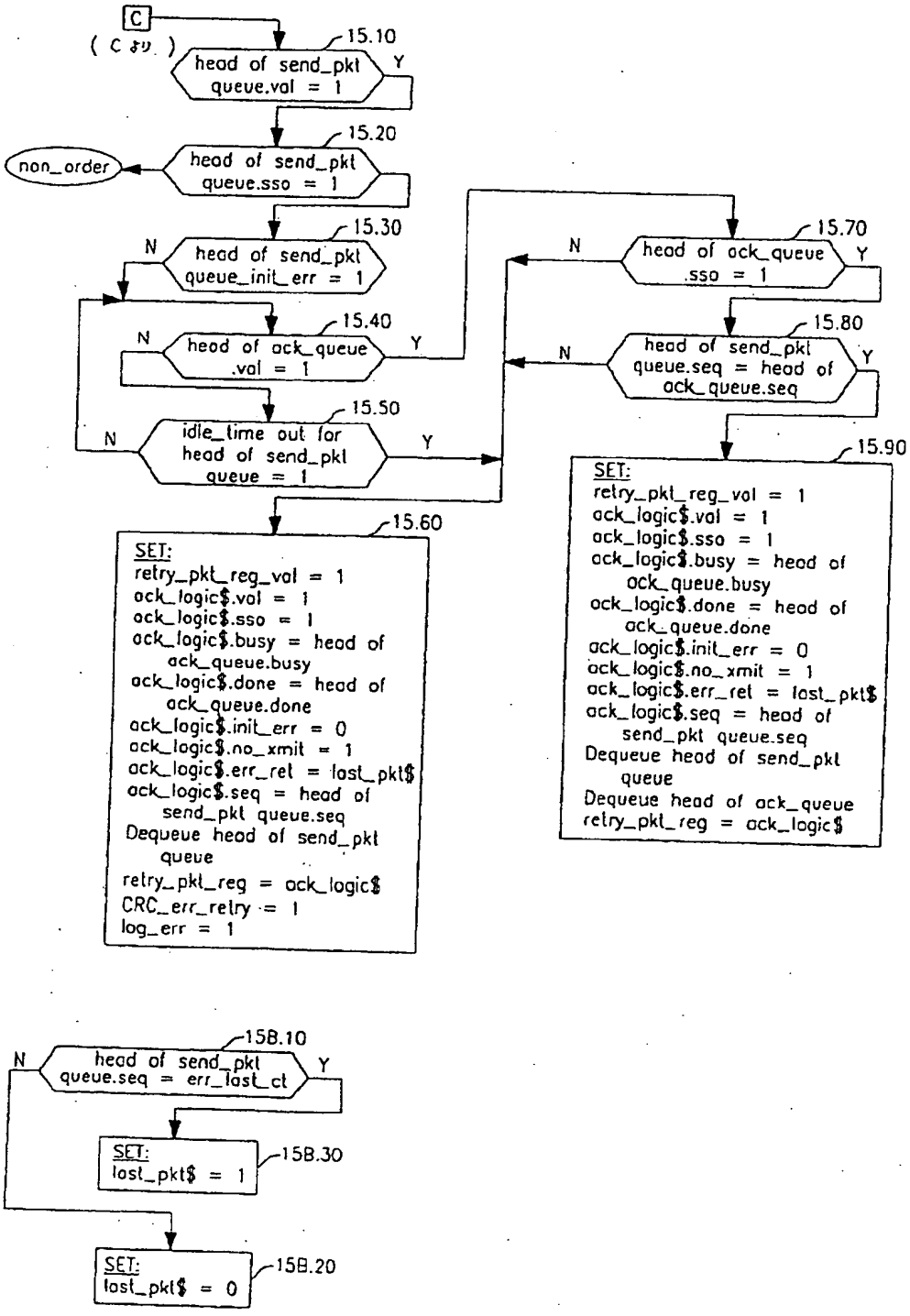
【図１４】


Ack 論理
B
13.10
head of send_pkt queue.vol = 1
Y
13.20
non_order
head of send_pkt queue.SSO = 1
Y
13.30
N
head of send_pkt queue.init_err = 1
Y
13.40
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 0
ack_logic$.no_xmit = 0
ack_logic$.err_ret = 1
ack_logic$.seq =
head of send_pkt queue.seq
Dequeue head of send_pkt queue
retry_pkt_reg = ack_logic$
13.50
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 1
ack_logic$.err_ret = 0
ack_logic$.seq =
head of send_pkt queue.seq
Dequeue head of send_pkt queue
CRC_err = 0
CRC_ack_chk = 1
retry_pkt_reg = ack_logic$

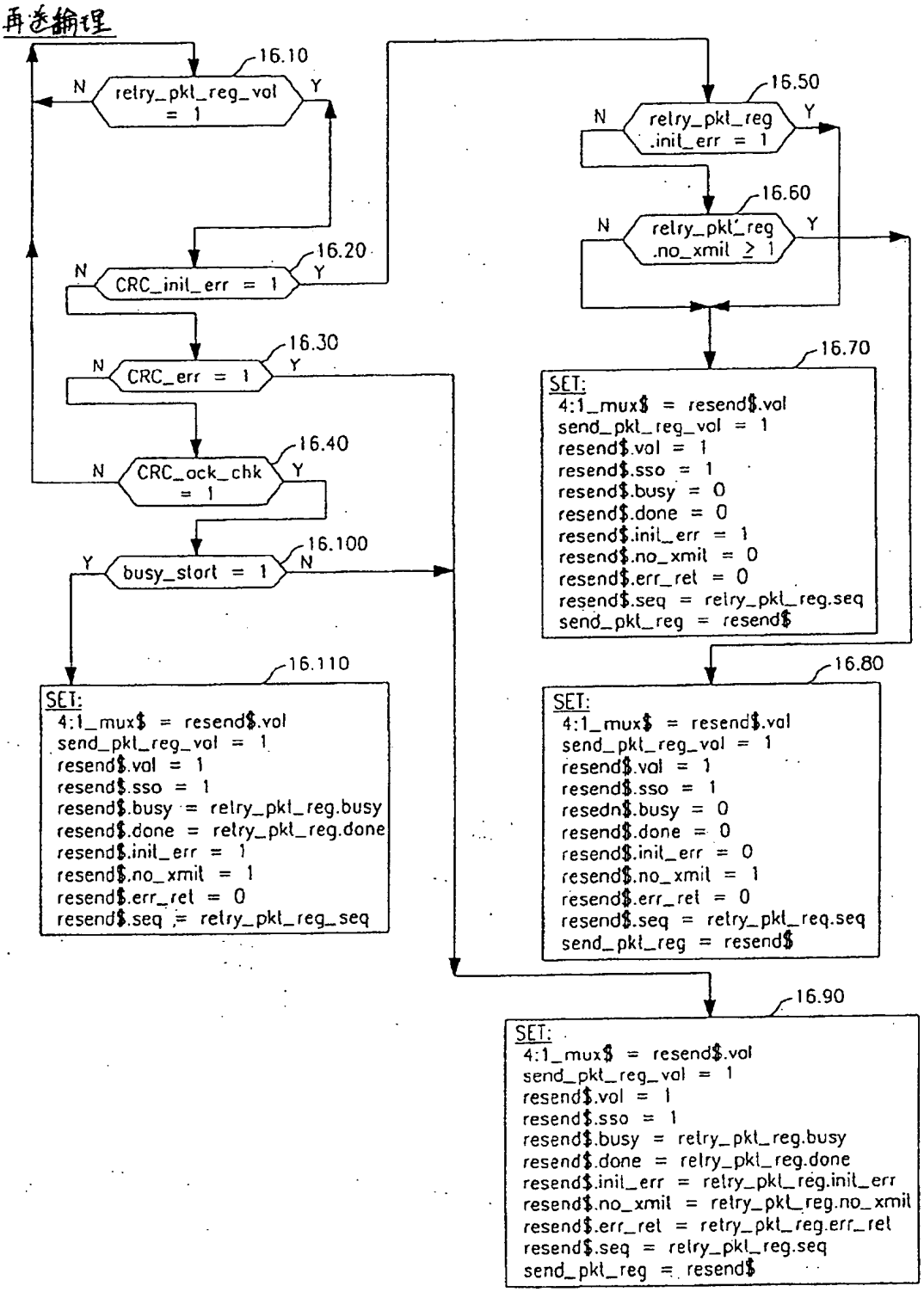
【図１５】


Ack 論理
/*Ack検査完了；エラーなし
C
(Cより)
14.10
head of send_pkt queue.val = 1
Y
14.20
N
non_order
head of send_pkt queue.SSO = 1
Y
14.30
head of send_pkt queue.init_err = 1
Y
(Fig. 22)
14.40
N
CRC_err_retry = 1
Y
14.50
SET:
CRC_err_retry = 0
CRC_ack_chk = 0
CRC_init_err = 1
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 0
ack_logic$.err_ret = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
retry_pkt_reg = ack_logic$
14.60
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = head of send_pkt queue.busy
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 1
ack_logic$.err_ret = 0
ack_logic$.seq = head of send_pkt queue.seq
dequeue head of ack_queue
CRC_ack_chk = 0
CRC_err_end = 1
busy_loop_snd = 0
retry_pkt_reg = ack_logic$

【図１６】

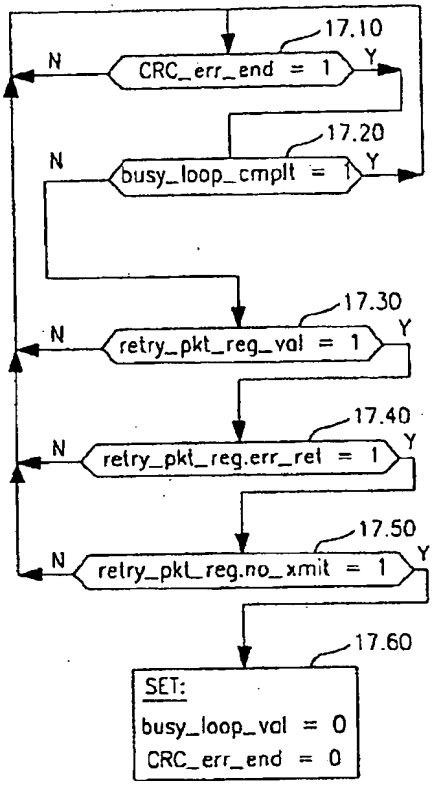

C
(C より)
15.10
head of send_pkt queue.val = 1
Y
15.20
non_order
head of send_pkt queue.sso = 1
15.30
N
head of send_pkt queue_init_err = 1
15.40
N
head of ack_queue .val = 1
Y
15.50
N
idle_time out for head of send_pkt queue = 1
Y
15.60
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = head of ack_queue.busy
ack_logic$.done = head of ack_queue.done
ack_logic$.init_err = 0
ack_logic$.no_xmit = 1
ack_logic$.err_ret = lost_pkt$
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
retry_pkt_reg = ack_logic$
CRC_err_retry = 1
log_err = 1
15.70
N
head of ack_queue .sso = 1
Y
15.80
N
head of send_pkt queue.seq = head of ack_queue.seq
Y
15.90
SET:
retry_pkt_reg_val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = head of ack_queue.busy
ack_logic$.done = head of ack_queue.done
ack_logic$.init_err = 0
ack_logic$.no_xmit = 1
ack_logic$.err_ret = lost_pkt$
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_pkt queue
Dequeue head of ack_queue
retry_pkt_reg = ack_logic$
15B.10
N
head of send_pkt queue.seq = err_lost_ct
Y
SET:
lost_pkt$ = 1
15B.30
SET:
lost_pkt$ = 0
15B.20

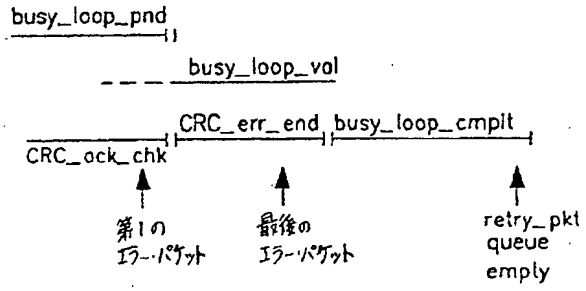
【図１７】


再送論理
16.10
retry_pkt_reg_val = 1
N
Y
16.20
CRC_init_err = 1
16.30
CRC_err = 1
16.40
CRC_ack_chk = 1
16.100
busy_start = 1
16.110
SET:
4:1_mux$ = resend$.val
send_pkt_reg_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = retry_pkt_reg.busy
resend$.done = retry_pkt_reg.done
resend$.init_err = 1
resend$.no_xmit = 1
resend$.err_rel = 0
resend$.seq = retry_pkt_reg_seq
16.50
retry_pkt_reg .init_err = 1
16.60
retry_pkt_reg .no_xmit ≥ 1
16.70
SET:
4:1_mux$ = resend$.val
send_pkt_reg_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = 0
resend$.done = 0
resend$.init_err = 1
resend$.no_xmit = 0
resend$.err_rel = 0
resend$.seq = retry_pkt_reg.seq
send_pkt_reg = resend$
16.80
SET:
4:1_mux$ = resend$.val
send_pkt_reg_val = 1
resend$.val = 1
resend$.sso = 1
resedn$.busy = 0
resend$.done = 0
resend$.init_err = 0
resend$.no_xmit = 1
resend$.err_rel = 0
resend$.seq = retry_pkt_reg.seq
send_pkt_reg = resend$
16.90
SET:
4:1_mux$ = resend$.val
send_pkt_reg_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = retry_pkt_reg.busy
resend$.done = retry_pkt_reg.done
resend$.init_err = retry_pkt_reg.init_err
resend$.no_xmit = retry_pkt_reg.no_xmit
resend$.err_rel = retry_pkt_reg.err_rel
resend$.seq = retry_pkt_reg.seq
send_pkt_reg = resend$

【図１８】

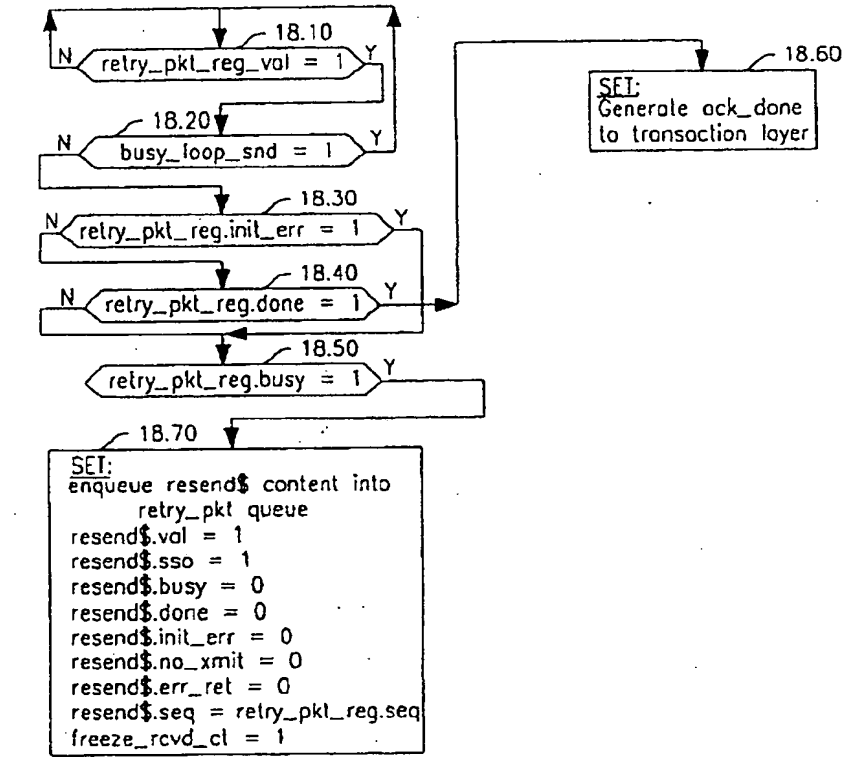
再送論理


17.10
N
CRC_err_end = 1
Y
17.20
N
busy_loop_cmplt = 1
Y
17.30
N
retry_pkt_reg_val = 1
Y
17.40
N
retry_pkt_reg.err_ret = 1
Y
17.50
N
retry_pkt_reg.no_xmit = 1
Y
17.60
SET:
busy_loop_val = 0
CRC_err_end = 0


busy_loop_pnd
busy_loop_val
CRC_err_end
busy_loop_cmplt
CRC_ock_chk
第1の
エラーパケット
最後の
エラーパケット
retry_pkt
queue
emply

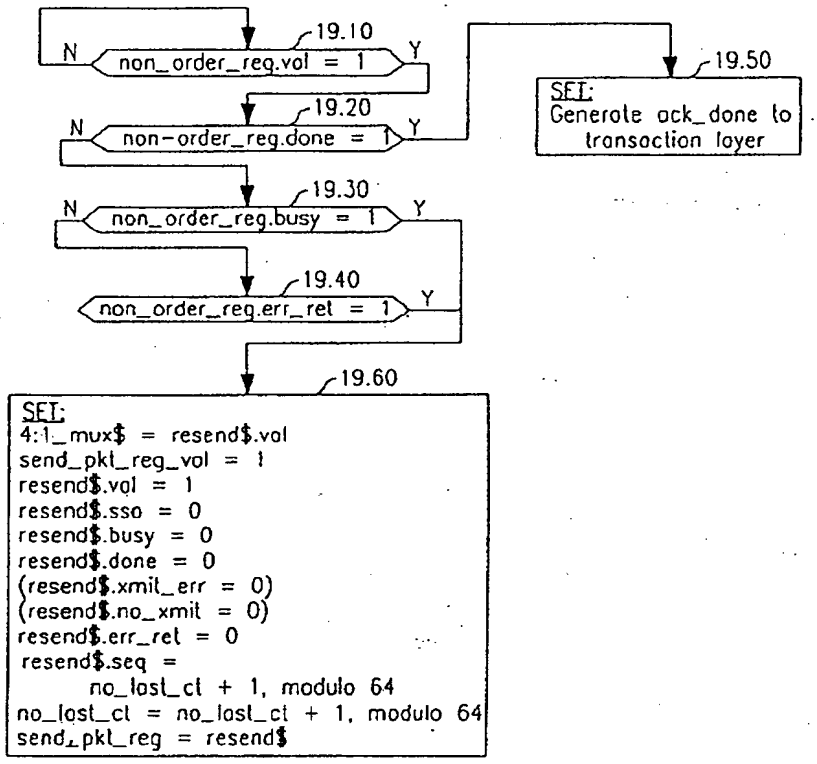
【図１９】

再送論理


18.10
N
retry_pkt_reg_val = 1
Y
18.20
N
busy_loop_snd = 1
Y
18.30
N
retry_pkt_reg.init_err = 1
Y
18.40
N
retry_pkt_reg.done = 1
Y
18.50
retry_pkt_reg.busy = 1
Y
18.70
SET:
enqueue resend$ content into
retry_pkt queue
resend$.val = 1
resend$.sso = 1
resend$.busy = 0
resend$.done = 0
resend$.init_err = 0
resend$.no_xmit = 0
resend$.err_ret = 0
resend$.seq = retry_pkt_reg.seq
freeze_rcvd_cl = 1
18.60
SET:
Generate ack_done
to transaction layer

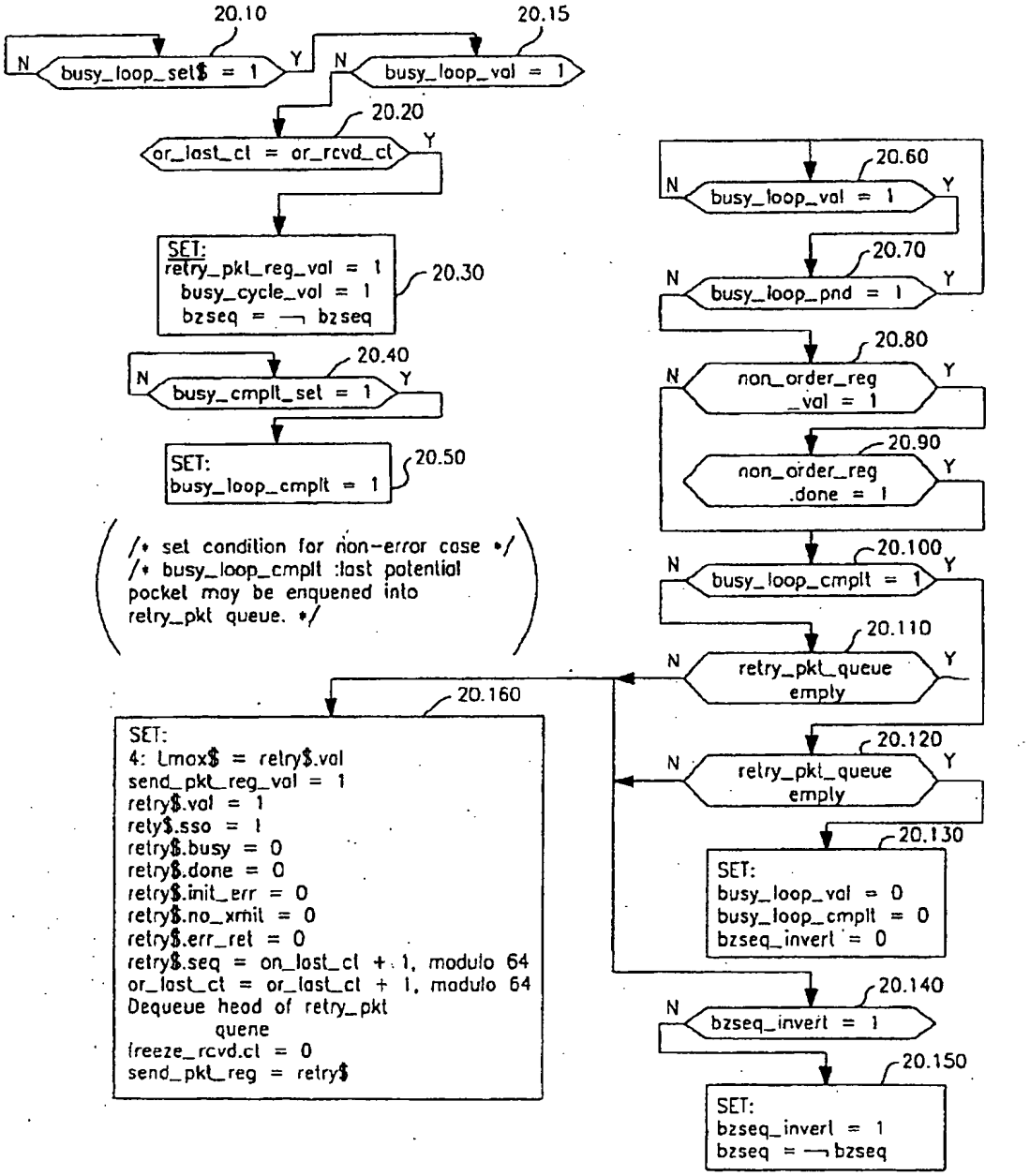
【図２０】

再送論理


19.10
N
non_order_req.vol = 1
Y
19.20
N
non-order_req.done = 1
Y
19.30
N
non_order_req.busy = 1
Y
19.40
non_order_req.err_ret = 1
Y
19.50
SET:
Generate ack_done to transaction layer
19.60
SET:
4:1_mux$ = resend$.vol
send_pkt_reg_vol = 1
resend$.vol = 1
resend$.sso = 0
resend$.busy = 0
resend$.done = 0
(resend$.xmit_err = 0)
(resend$.no_xmit = 0)
resend$.err_ret = 0
resend$.seq =
no_last_ct + 1, modulo 64
no_last_ct = no_last_ct + 1, modulo 64
send_pkt_reg = resend$

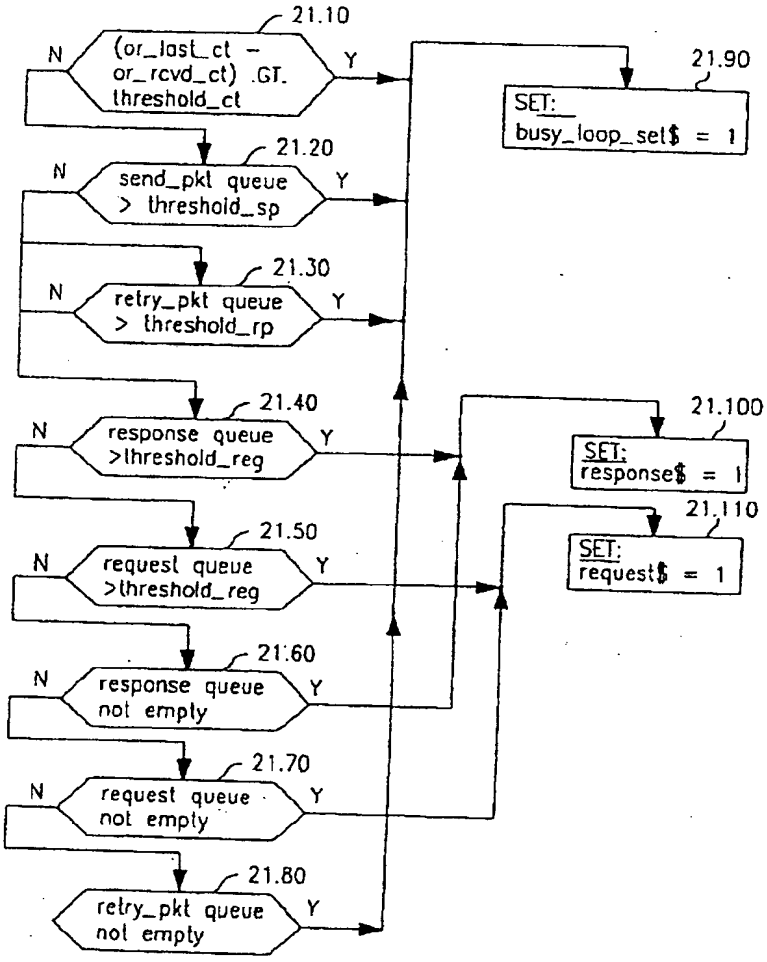
【図２１】

再試行論理

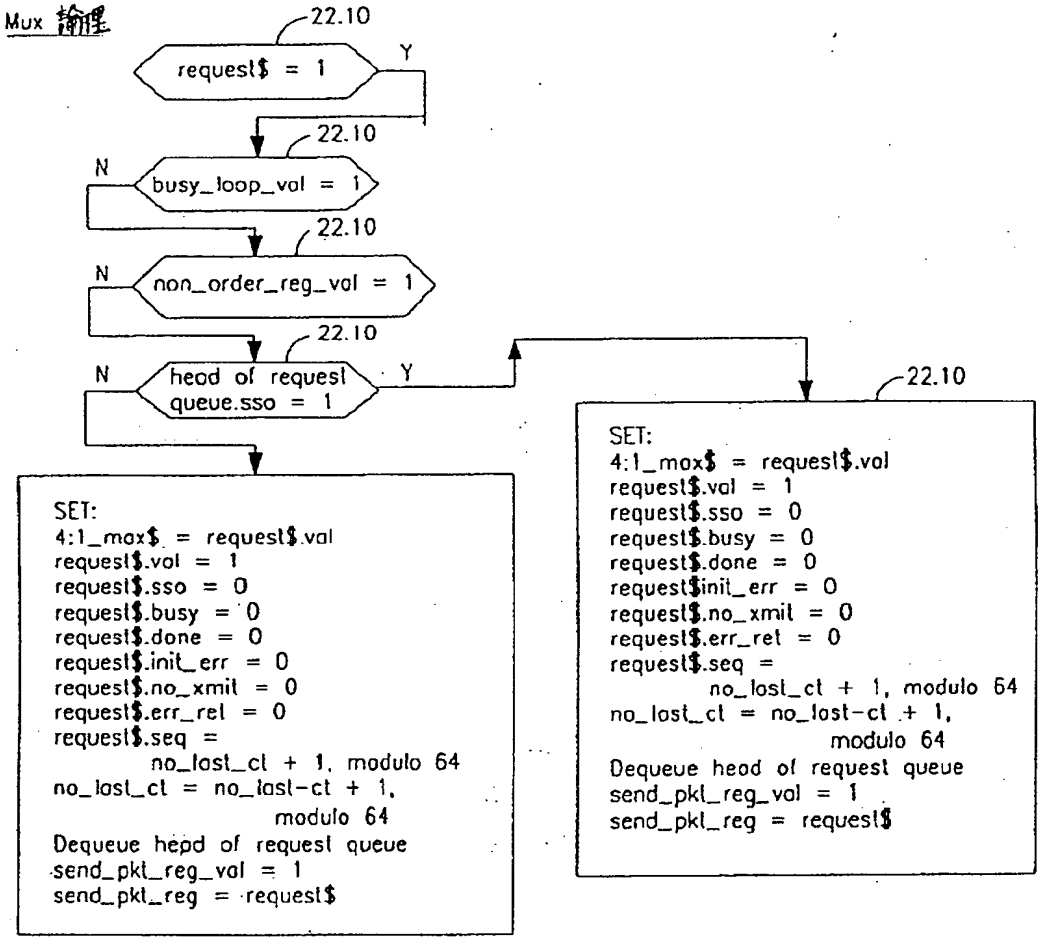

20.10
N busy_loop_set$ = 1 Y
20.15
N busy_loop_val = 1
20.20
or_last_ct = or_rcvd_ct Y
SET:
retry_pkt_reg_val = 1
busy_cycle_val = 1
bzseq = ¬ bzseq
20.30
20.40
N busy_cmplt_set = 1 Y
SET:
busy_loop_cmplt = 1
20.50
/* set condition for non-error case */
/* busy_loop_cmplt :last potential
pocket may be enqueued into
retry_pkt queue. */
20.60
N busy_loop_val = 1 Y
20.70
N busy_loop_pnd = 1 Y
20.80
N non_order_reg
_val = 1 Y
20.90
non_order_reg
.done = 1 Y
20.100
N busy_loop_cmplt = 1 Y
20.110
N retry_pkt_queue
empty Y
20.120
N retry_pkt_queue
empty Y
20.130
SET:
busy_loop_val = 0
busy_loop_cmplt = 0
bzseq_invert = 0
20.140
N bzseq_invert = 1
20.150
SET:
bzseq_invert = 1
bzseq = ¬ bzseq
20.160
SET:
4: tmox$ = retry$.val
send_pkt_reg_val = 1
retry$.val = 1
rety$.sso = 1
retry$.busy = 0
retry$.done = 0
retry$.init_err = 0
retry$.no_xmit = 0
retry$.err_ret = 0
retry$.seq = on_last_ct + 1, modulo 64
or_last_ct = or_last_ct + 1, modulo 64
Dequeue head of retry_pkt
quene
freeze_rcvd.ct = 0
send_pkt_reg = retry$

【図２２】

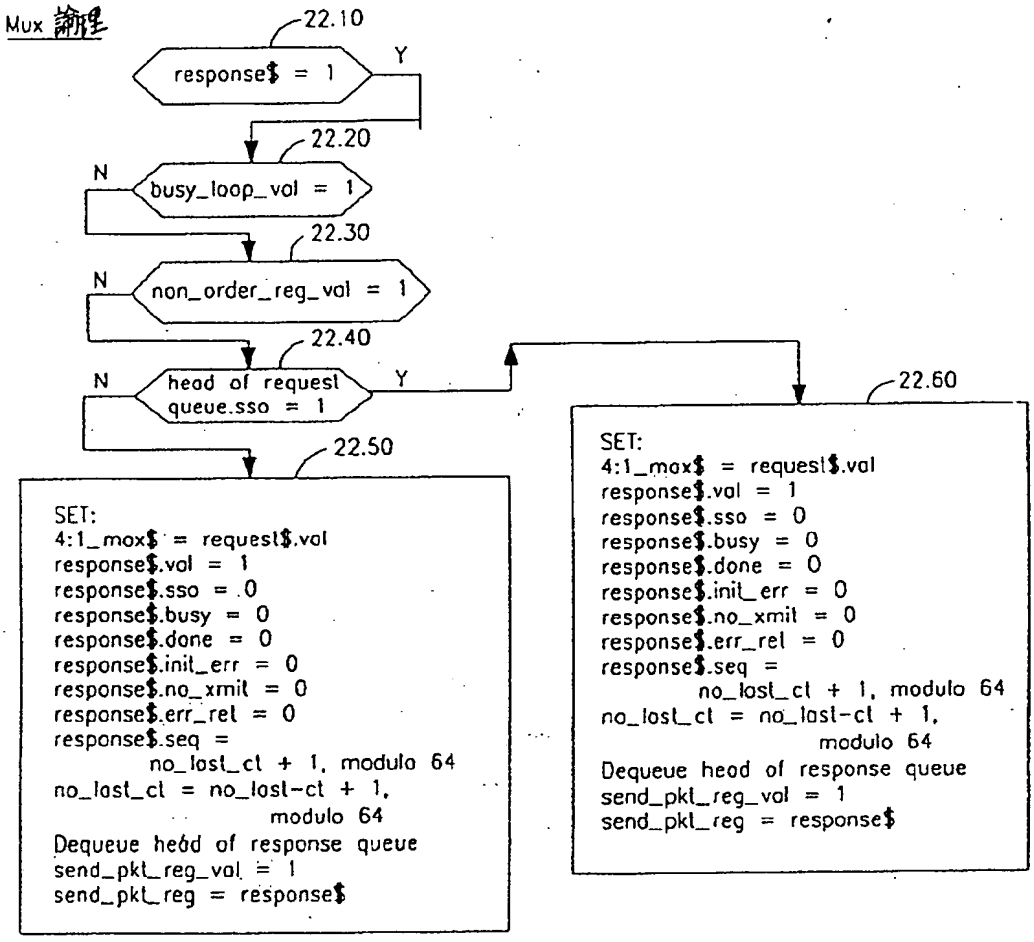
Mux 論理


21.10
(or_last_ct - or_rcvd_ct) .GT. threshold_ct
N
Y
21.90
SET: busy_loop_set$ = 1
21.20
send_pkt queue > threshold_sp
21.30
retry_pkt queue > threshold_rp
21.40
response queue >threshold_reg
21.100
SET: response$ = 1
21.50
request queue >threshold_reg
21.110
SET: request$ = 1
21.60
response queue not empty
21.70
request queue not empty
21.80
retry_pkt queue not empty

【図２３】

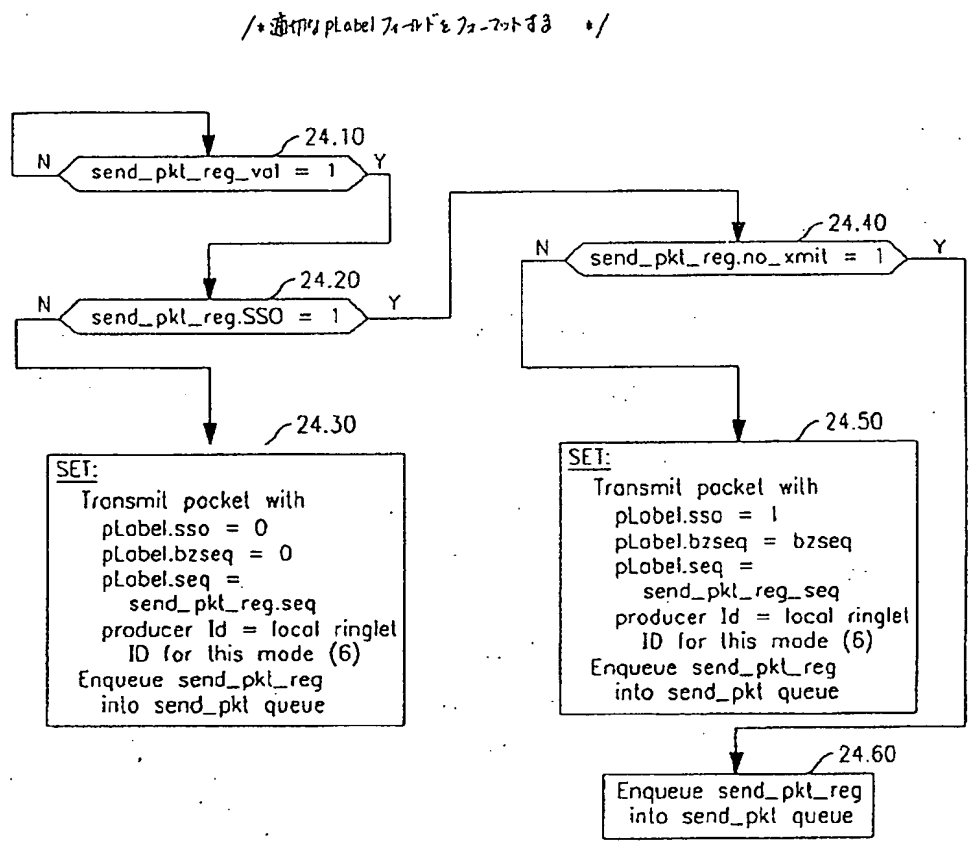

Mux 論理
22.10
request$ = 1
Y
22.10
N
busy_loop_val = 1
22.10
N
non_order_reg_val = 1
22.10
N
head of request queue.sso = 1
Y
22.10
SET:
4:1_mox$ = request$.val
request$.val = 1
request$.sso = 0
request$.busy = 0
request$.done = 0
request$init_err = 0
request$.no_xmit = 0
request$.err_ret = 0
request$.seq =
no_lost_ct + 1, modulo 64
no_lost_ct = no_lost-ct + 1,
modulo 64
Dequeue head of request queue
send_pkt_reg_val = 1
send_pkt_reg = request$
SET:
4:1_mox$ = request$.val
request$.val = 1
request$.sso = 0
request$.busy = 0
request$.done = 0
request$init_err = 0
request$.no_xmit = 0
request$.err_ret = 0
request$.seq =
no_lost_ct + 1, modulo 64
no_lost_ct = no_lost-ct + 1,
modulo 64
Dequeue head of request queue
send_pkt_reg_val = 1
send_pkt_reg = request$

【図２４】


Mux 論理
22.10
response$ = 1
Y
22.20
N
busy_loop_val = 1
22.30
N
non_order_reg_val = 1
22.40
N
head of request queue.sso = 1
Y
22.50
SET:
4:1_mox$ = request$.val
response$.val = 1
response$.sso = 0
response$.busy = 0
response$.done = 0
response$.init_err = 0
response$.no_xmit = 0
response$.err_rel = 0
response$.seq =
no_lost_ct + 1, modulo 64
no_lost_ct = no_lost-ct + 1,
modulo 64
Dequeue head of response queue
send_pkt_reg_val = 1
send_pkt_reg = response$
22.60
SET:
4:1_mox$ = request$.val
response$.val = 1
response$.sso = 0
response$.busy = 0
response$.done = 0
response$.init_err = 0
response$.no_xmit = 0
response$.err_rel = 0
response$.seq =
no_lost_ct + 1, modulo 64
no_lost_ct = no_lost-ct + 1,
modulo 64
Dequeue head of response queue
send_pkt_reg_val = 1
send_pkt_reg = response$

【図２５】

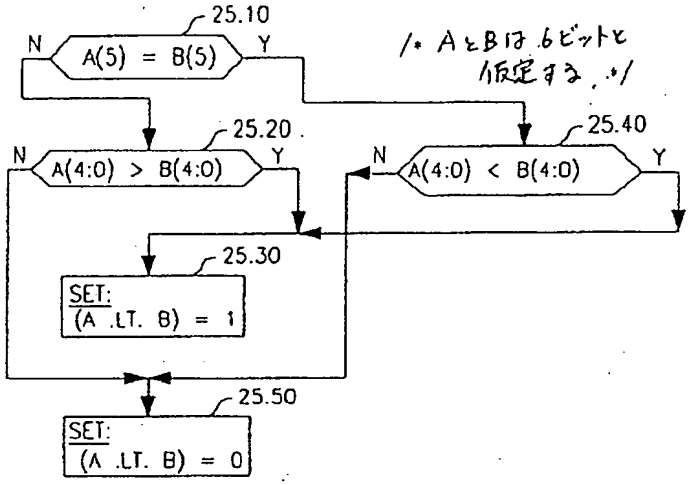
CRC 生成論理


/* 適切な pLabel フィールドをフォーマットする */
24.10
N
send_pkt_reg_val = 1
Y
24.20
N
send_pkt_reg.SSO = 1
Y
24.30
SET:
Transmit packet with
pLabel.sso = 0
pLabel.bzseq = 0
pLabel.seq =
send_pkt_reg.seq
producer Id = local ringlet
ID for this mode (6)
Enqueue send_pkt_reg
into send_pkt queue
24.40
N
send_pkt_reg.no_xmit = 1
Y
24.50
SET:
Transmit packet with
pLabel.sso = 1
pLabel.bzseq = bzseq
pLabel.seq =
send_pkt_reg_seq
producer Id = local ringlet
ID for this mode (6)
Enqueue send_pkt_reg
into send_pkt queue
24.60
Enqueue send_pkt_reg
into send_pkt queue

【図２６】

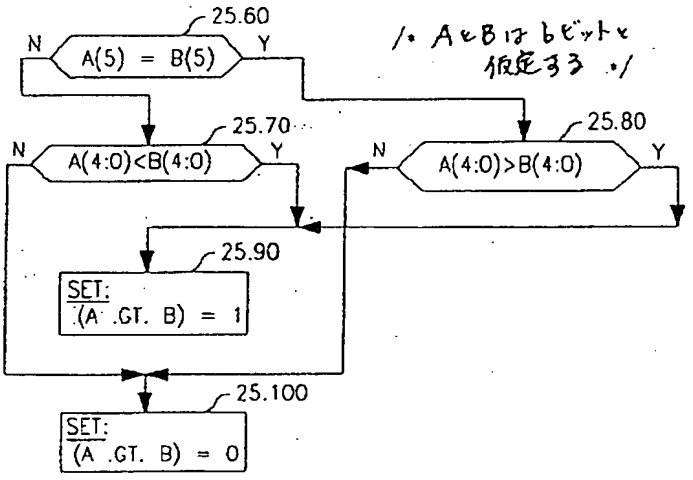
比較論理

A .LT. B



A

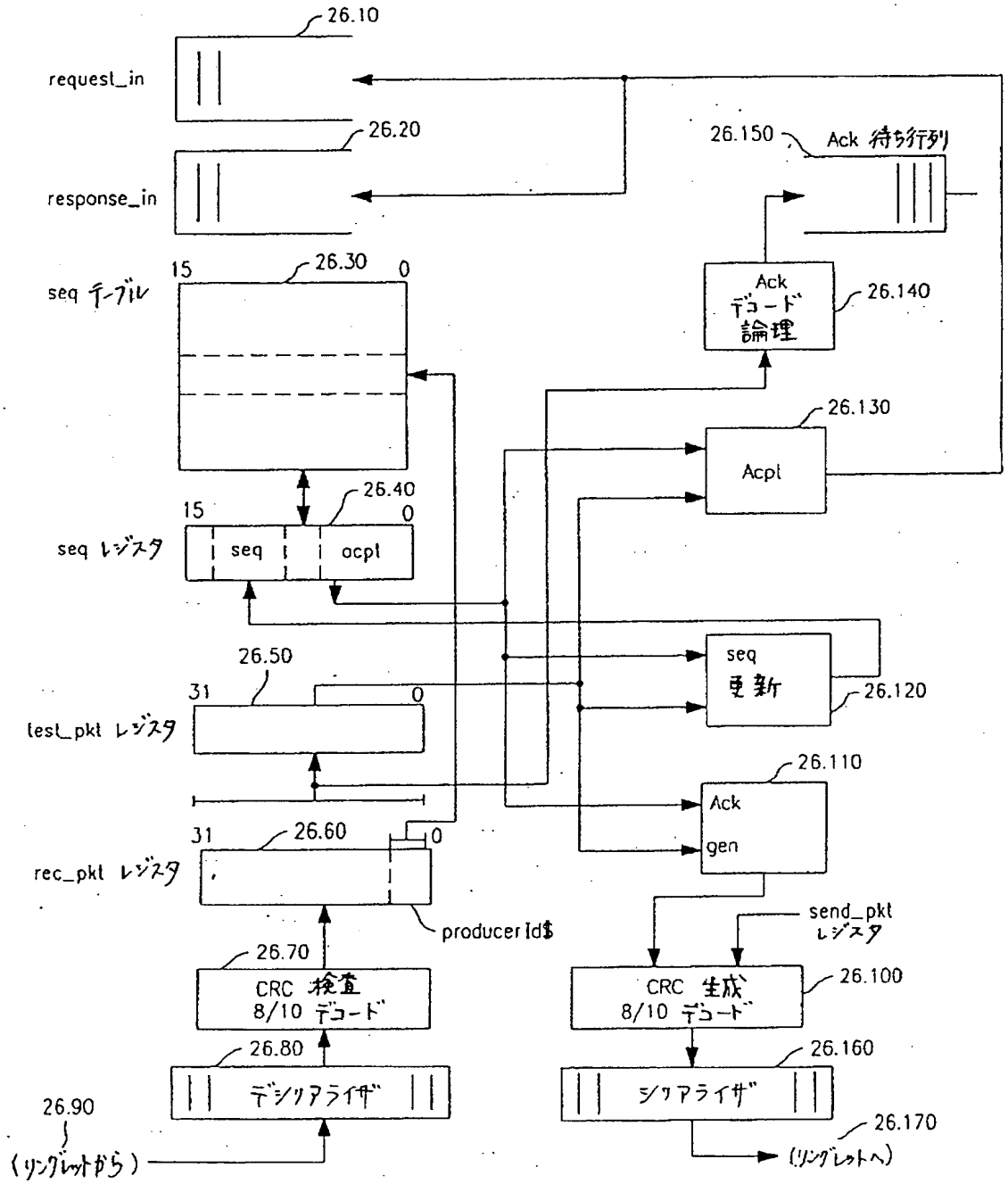
A .GT. B



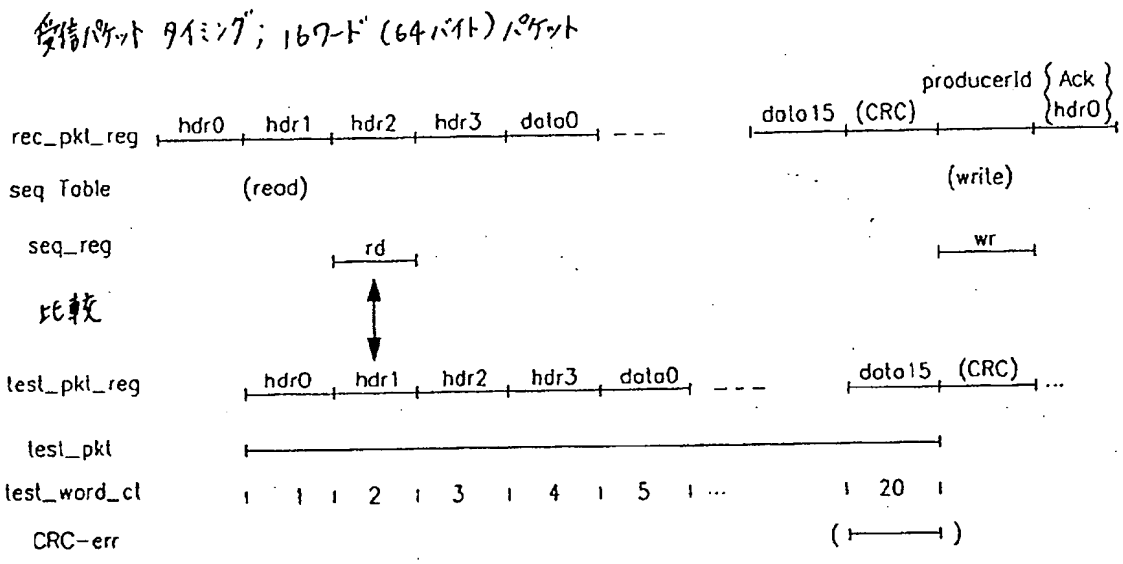
B

【図２７】

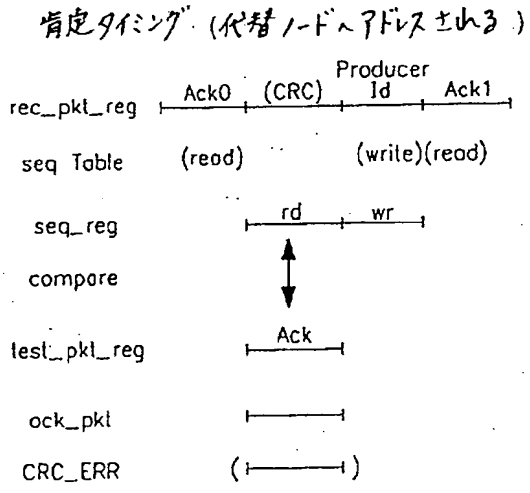
受信ブロック図


26.10
request_in
26.20
response_in
26.150
Ack 待ち行列
15
26.30
0
seq テーブル
Ack
デコード
論理
26.140
26.130
Acpt
26.40
15
0
seq レジスタ
seq
acpt
26.50
seq
更新
26.120
31
0
test_pkt レジスタ
26.110
Ack
gen
31
26.60
0
rec_pkt レジスタ
producer Id$
send_pkt
レジスタ
26.70
CRC 検査
8/10 デコード
CRC 生成
8/10 デコード
26.100
26.80
デシリアライザ
26.160
シリアライザ
26.90
(リングレットから)
26.170
(リングレットへ)

【図２８】


受信パケット タイミング; 16ワード (64バイト) パケット
rec_pkt_reg
hdr0
hdr1
hdr2
hdr3
data0
data15
(CRC)
producerId
Ack
hdr0
seq Table
(read)
(write)
seq_reg
rd
wr
比較
test_pkt_reg
test_pkt
test_word_cl
1
2
3
4
5
20
CRC-err

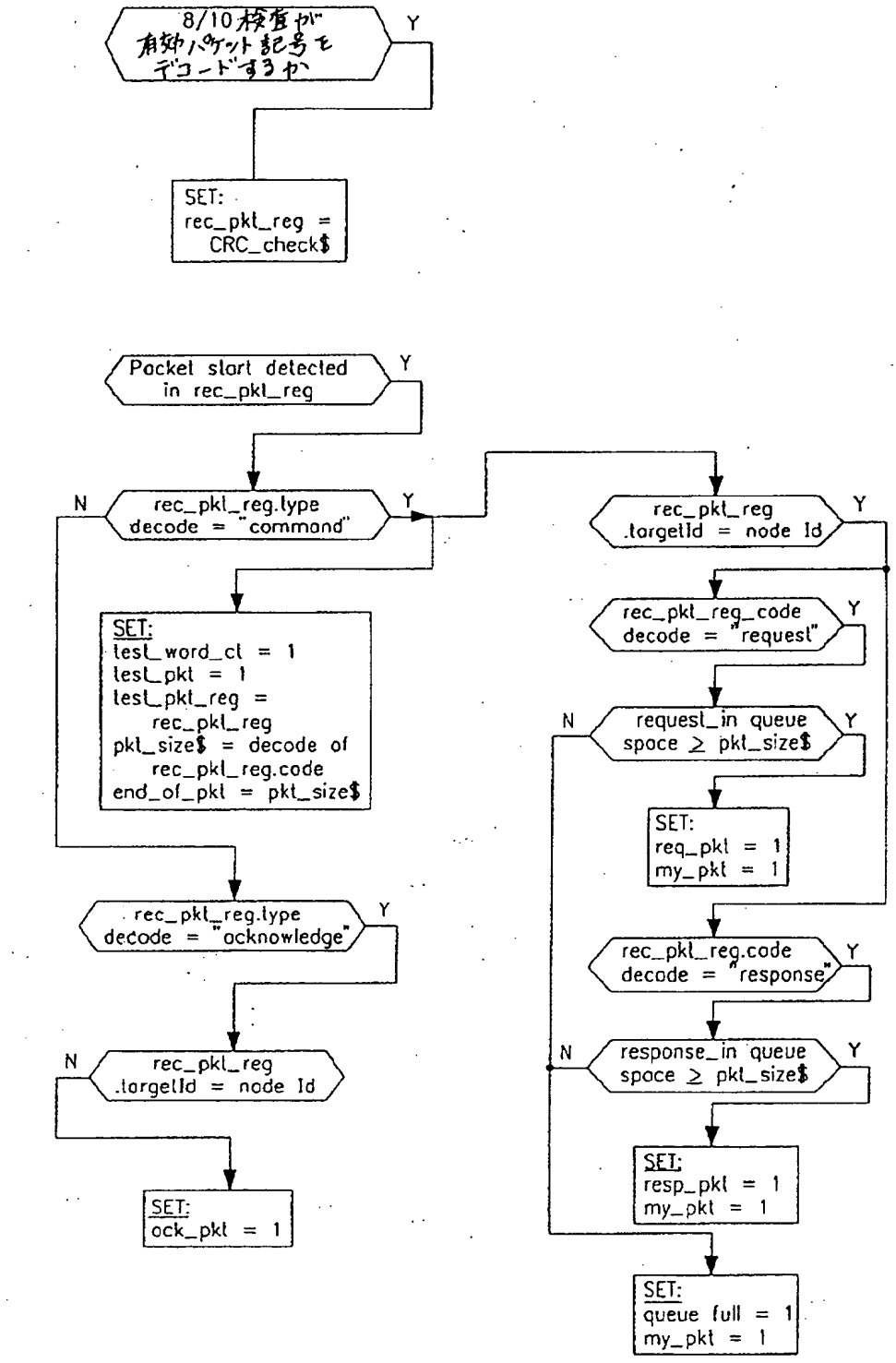
A


肯定タイミング (代替ノードへアドレス立ち上る)
rec_pkt_reg
Ack0
(CRC)
Producer Id
Ack1
seq Table
(read)
(write)(read)
seq_reg
rd
wr
compare
test_pkt_reg
Ack
ock_pkt
CRC_ERR

B

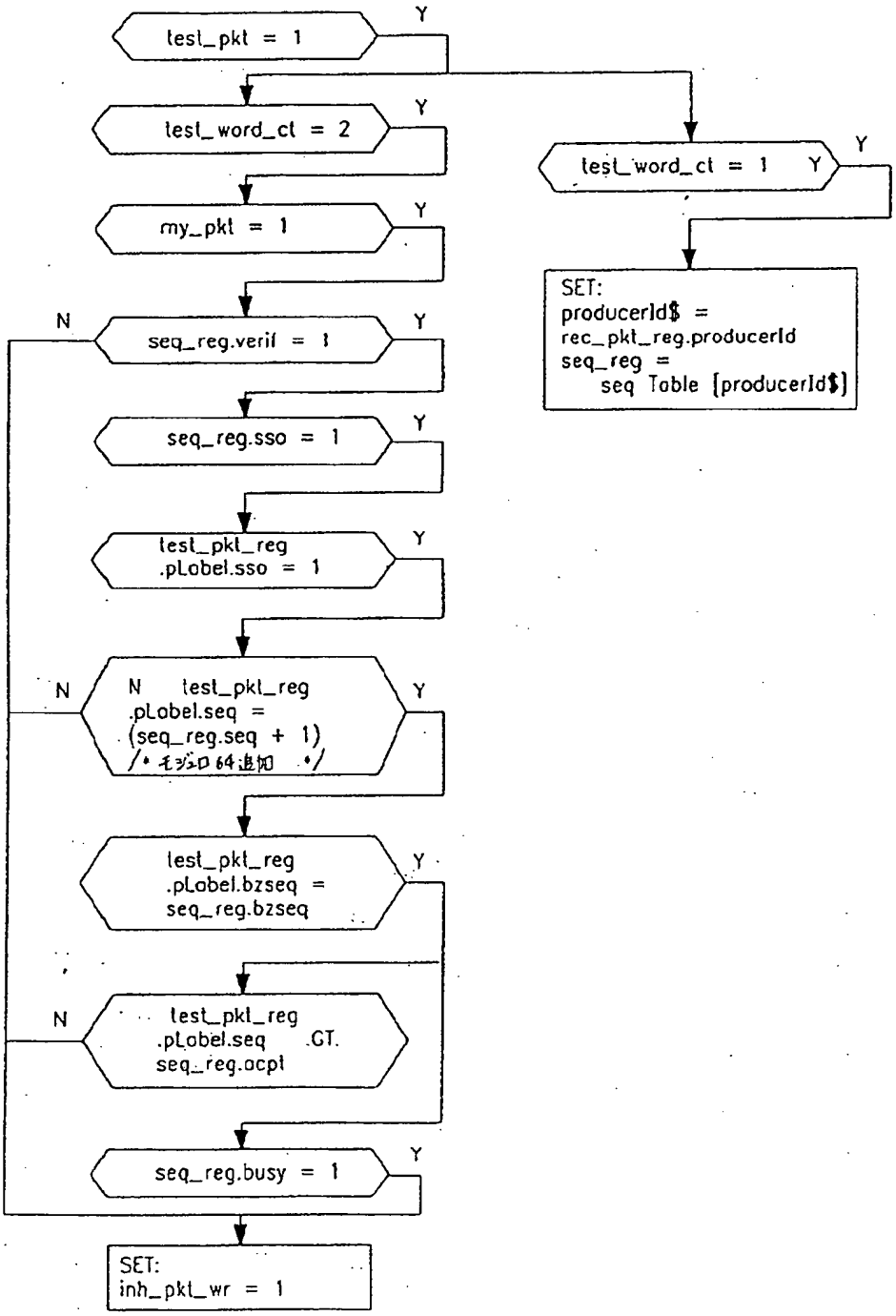
【図２９】

受け論理


8/10検査が
有効パケット記号を
デコードするか
Y
SET:
rec_pkt_reg =
CRC_check$
Packet start detected
in rec_pkt_reg
Y
N
rec_pkt_reg.type
decode = "command"
Y
SET:
test_word_cl = 1
test_pkt = 1
test_pkt_reg =
rec_pkt_reg
pkt_size$ = decode of
rec_pkt_reg.code
end_of_pkt = pkt_size$
rec_pkt_reg.type
decode = "acknowledge"
Y
N
rec_pkt_reg
.targetId = node Id
SET:
ack_pkt = 1
rec_pkt_reg
.targetId = node Id
Y
rec_pkt_reg_code
decode = "request"
Y
N
request_in queue
space ≥ pkt_size$
Y
SET:
req_pkt = 1
my_pkt = 1
rec_pkt_reg.code
decode = "response"
Y
N
response_in queue
space ≥ pkt_size$
Y
SET:
resp_pkt = 1
my_pkt = 1
SET:
queue full = 1
my_pkt = 1

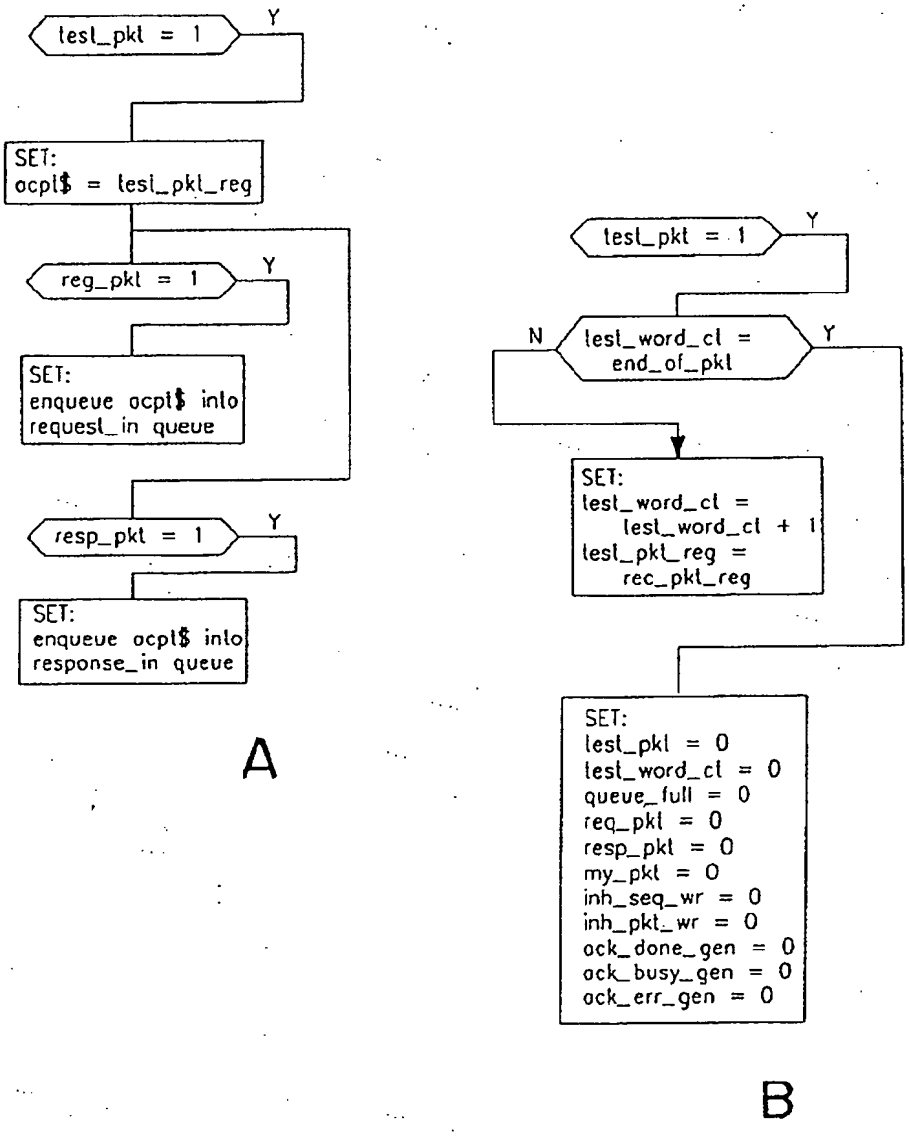
【図３０】

受入れ論理


test_pkt = 1
Y
test_word_ct = 2
Y
my_pkt = 1
Y
seq_reg.veril = 1
Y
N
seq_reg.sso = 1
Y
test_pkt_reg
.plabel.sso = 1
Y
N test_pkt_reg
.plabel.seq =
(seq_reg.seq + 1)
/* モジュロ64進加 */
N
Y
test_pkt_reg
.plabel.bzseq =
seq_reg.bzseq
Y
test_pkt_reg
.plabel.seq .GT.
seq_reg.ocpl
N
seq_reg.busy = 1
Y
SET:
inh_pkt_wr = 1
test_word_ct = 1
Y
Y
SET:
producerId$ =
rec_pkt_reg.producerId
seq_reg =
seq Table [producerId$]

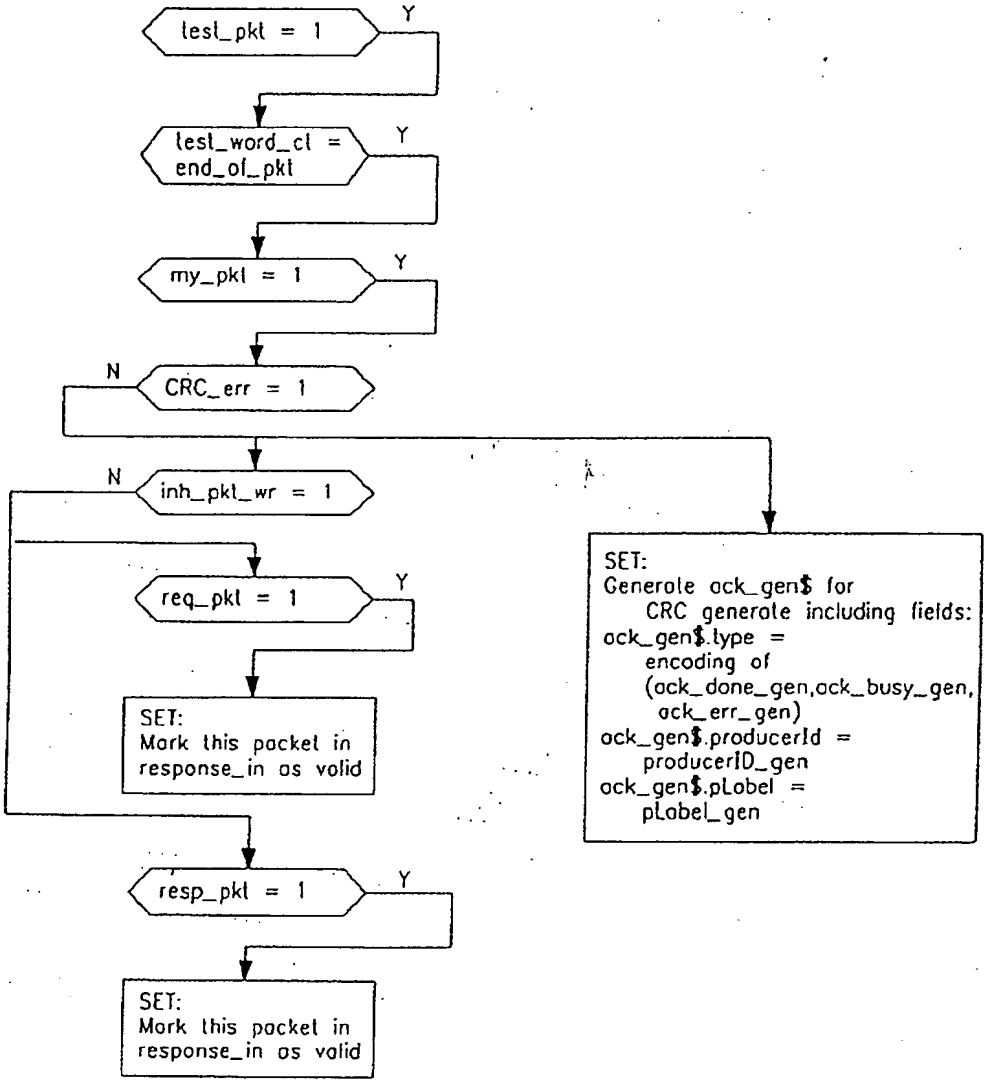
【図３１】

受入れ及びack生成論理



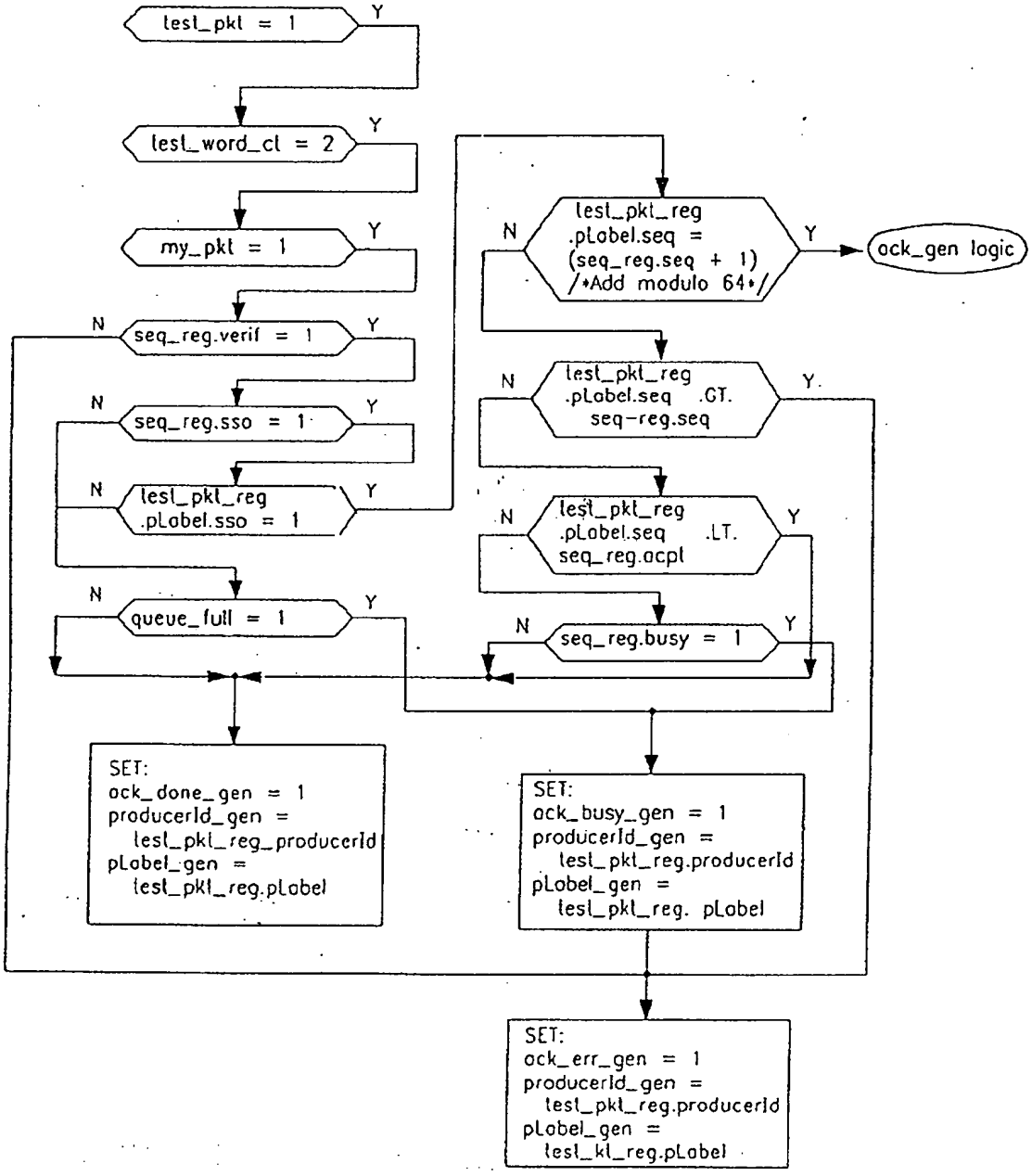
【図３２】

受入及びack生成論理



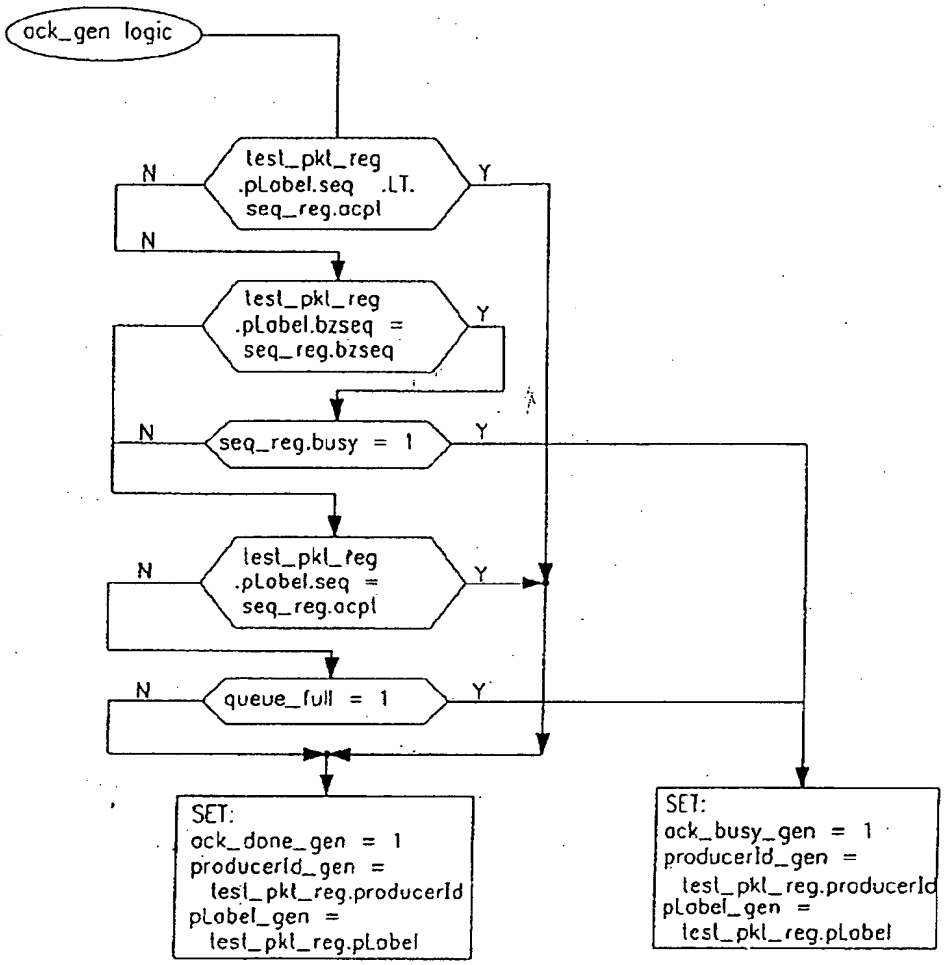
【図３３】

ack 生成論理

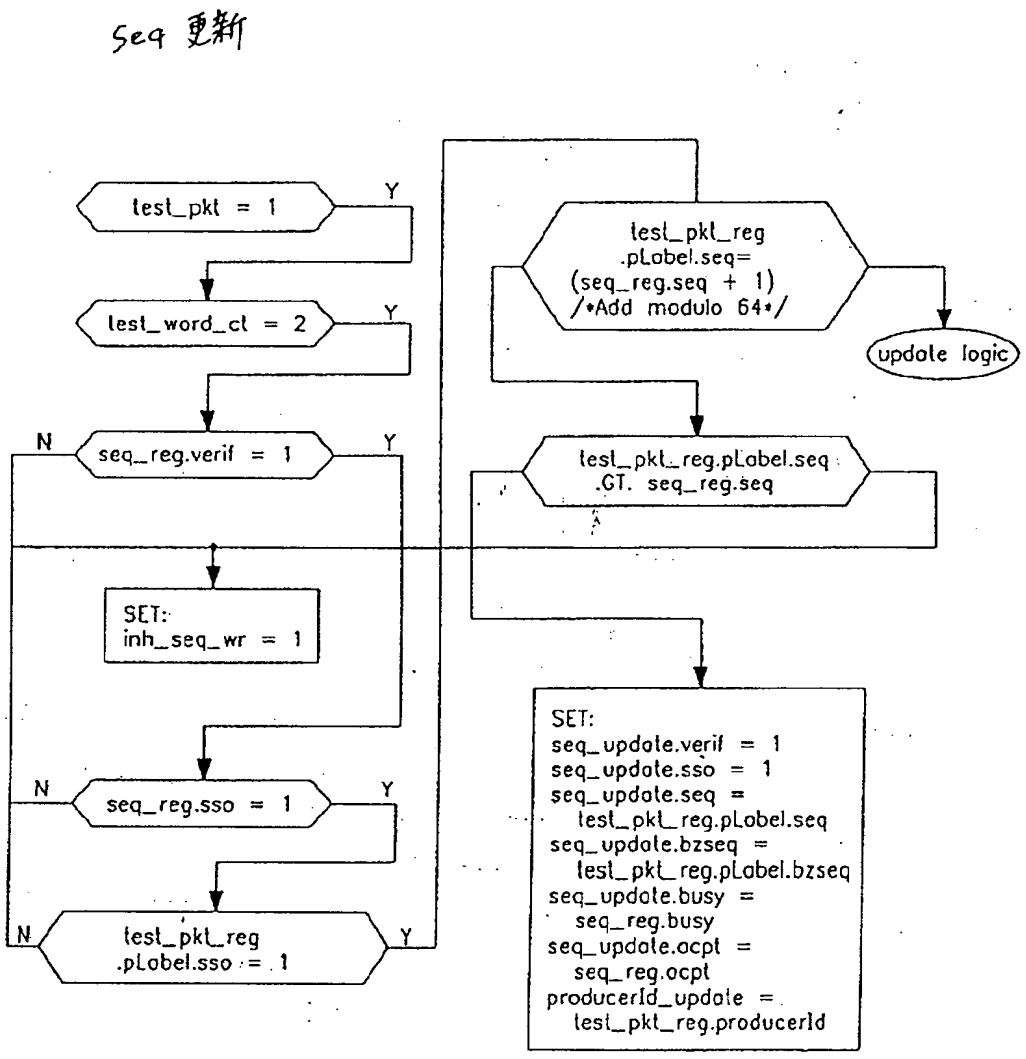

test_pkt = 1
Y
test_word_cl = 2
Y
my_pkt = 1
Y
N
seq_reg.verif = 1
Y
N
seq_reg.sso = 1
Y
N
test_pkt_reg
.pLabel.sso = 1
Y
N
queue_full = 1
Y
SET:
ack_done_gen = 1
producerId_gen =
test_pkt_reg_producerId
pLabel_gen =
test_pkt_reg.pLabel
test_pkt_reg
.pLabel.seq =
(seq_reg.seq + 1)
/*Add modulo 64*/
N
Y
ack_gen logic
N
test_pkt_reg
.pLabel.seq .GT.
seq-reg.seq
Y
N
test_pkt_reg
.pLabel.seq .LT.
seq_reg.acpt
Y
N
seq_reg.busy = 1
Y
SET:
ack_busy_gen = 1
producerId_gen =
test_pkt_reg.producerId
pLabel_gen =
test_pkt_reg. pLabel
SET:
ack_err_gen = 1
producerId_gen =
test_pkt_reg.producerId
pLabel_gen =
test_kt_reg.pLabel

【図３４】

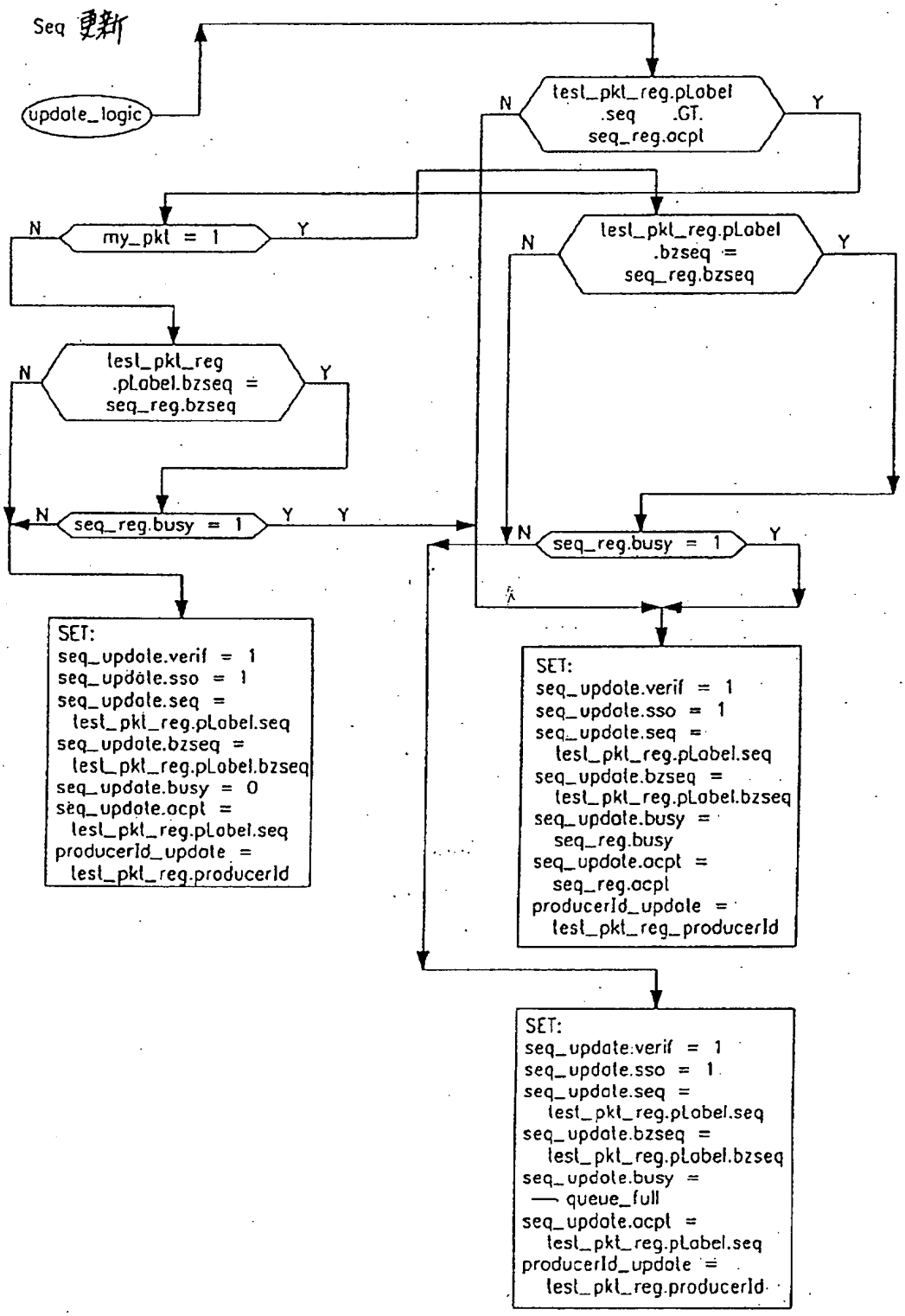
ack生成論理


ack_gen logic
test_pkt_reg .pLabel.seq .LT. seq_reg.acpl
N
Y
test_pkt_reg .pLabel.bzseq = seq_reg.bzseq
seq_reg.busy = 1
test_pkt_reg .pLabel.seq = seq_reg.acpl
queue_full = 1
SET:
ack_done_gen = 1
producerId_gen = test_pkt_reg.producerId
pLabel_gen = test_pkt_reg.pLabel
SET:
ack_busy_gen = 1
producerId_gen = test_pkt_reg.producerId
pLabel_gen = test_pkt_reg.pLabel

【図３５】

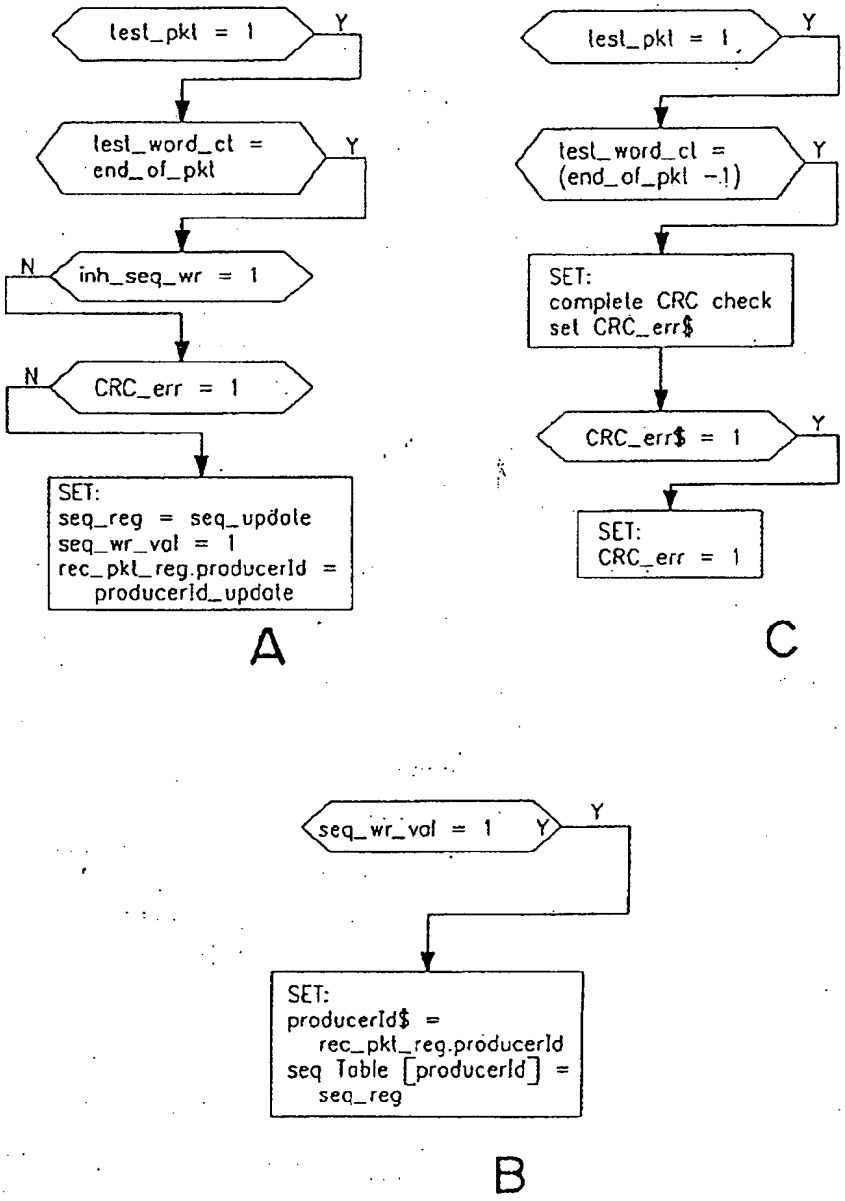

Seq 更新
test_pkt = 1
Y
test_word_ct = 2
Y
N
seq_reg.verif = 1
Y
SET:
inh_seq_wr = 1
N
seq_reg.sso = 1
Y
N
test_pkt_reg
.pLabel.sso = 1
Y
test_pkt_reg
.pLabel.seq=
(seq_reg.seq + 1)
/*Add modulo 64*/
update logic
test_pkt_reg.pLabel.seq
.GT. seq_reg.seq
SET:
seq_update.verif = 1
seq_update.sso = 1
seq_update.seq =
test_pkt_reg.pLabel.seq
seq_update.bzseq =
test_pkt_reg.pLabel.bzseq
seq_update.busy =
seq_reg.busy
seq_update.ocpt =
seq_reg.ocpt
producerId_update =
test_pkt_reg.producerId

【図３６】


Seq 更新
update_logic
test_pkt_reg.pLabel .seq .GT. seq_reg.ocpl
N
Y
my_pkt = 1
test_pkt_reg.pLabel .bzseq = seq_reg.bzseq
test_pkt_reg .pLabel.bzseq = seq_reg.bzseq
seq_reg.busy = 1
SET:
seq_update.verif = 1
seq_update.sso = 1
seq_update.seq = test_pkt_reg.pLabel.seq
seq_update.bzseq = test_pkt_reg.pLabel.bzseq
seq_update.busy = 0
seq_update.ocpl = test_pkt_reg.pLabel.seq
producerId_update = test_pkt_reg.producerId
SET:
seq_update.verif = 1
seq_update.sso = 1
seq_update.seq = test_pkt_reg.pLabel.seq
seq_update.bzseq = test_pkt_reg.pLabel.bzseq
seq_update.busy = seq_reg.busy
seq_update.ocpl = seq_reg.ocpl
producerId_update = test_pkt_reg_producerId
SET:
seq_update.verif = 1
seq_update.sso = 1
seq_update.seq = test_pkt_reg.pLabel.seq
seq_update.bzseq = test_pkt_reg.pLabel.bzseq
seq_update.busy = queue_full
seq_update.ocpl = test_pkt_reg.pLabel.seq
producerId_update = test_pkt_reg.producerId

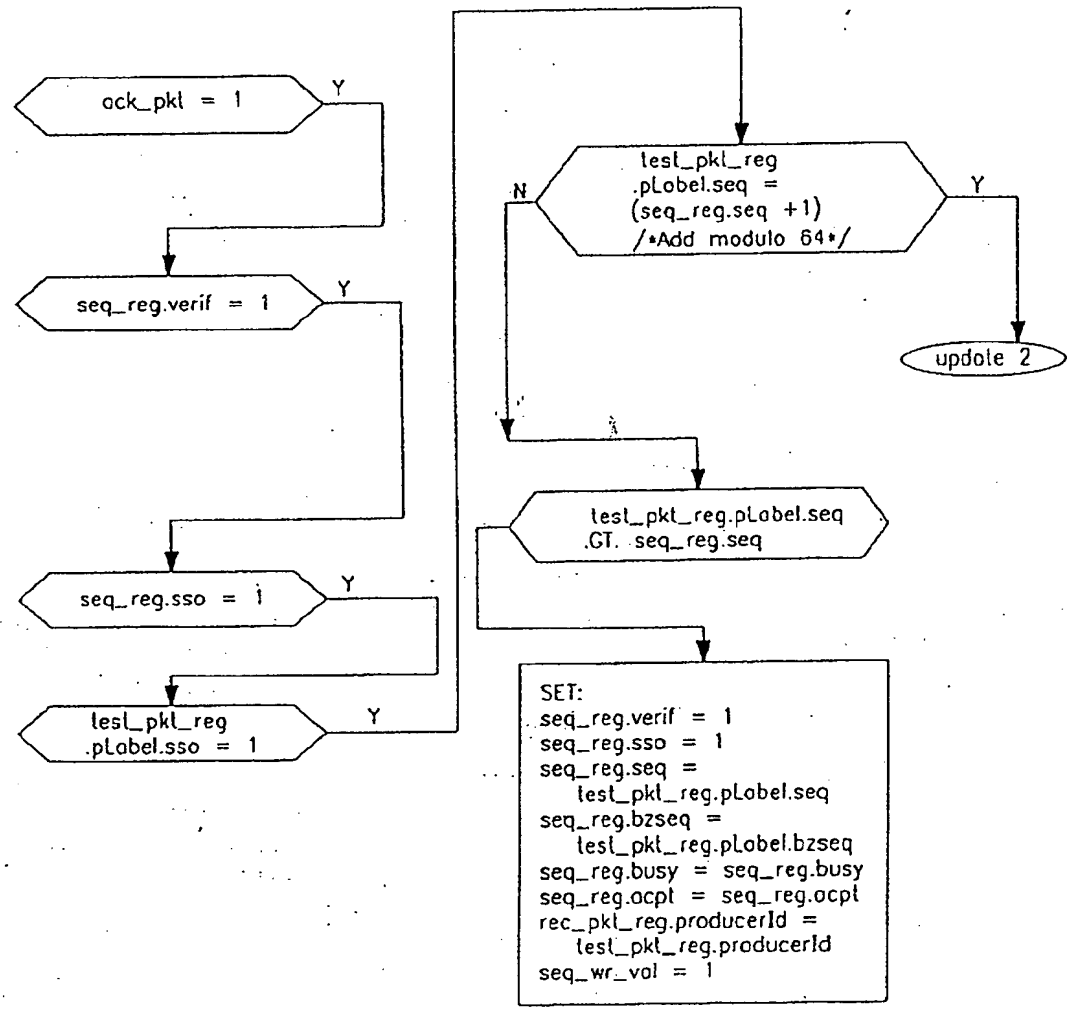
【図３７】

Seq 更新

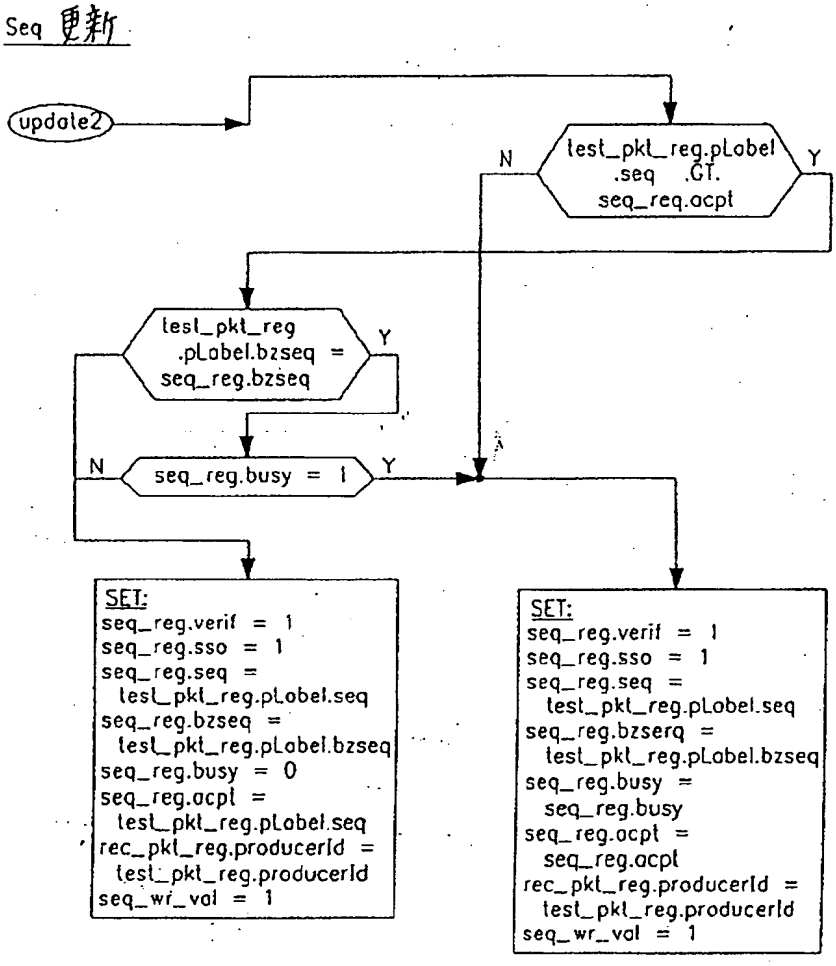

test_pkt = 1
Y
test_word_ct = end_of_pkt
Y
N
inh_seq_wr = 1
N
CRC_err = 1
SET:
seq_reg = seq_update
seq_wr_val = 1
rec_pkt_reg.producerId = producerId_update
A
test_pkt = 1
Y
test_word_ct = (end_of_pkt -1)
Y
SET:
complete CRC check
set CRC_err$
CRC_err$ = 1
Y
SET:
CRC_err = 1
C
seq_wr_val = 1
Y
SET:
producerId$ = rec_pkt_reg.producerId
seq Table [producerId] = seq_reg
B

【図３８】

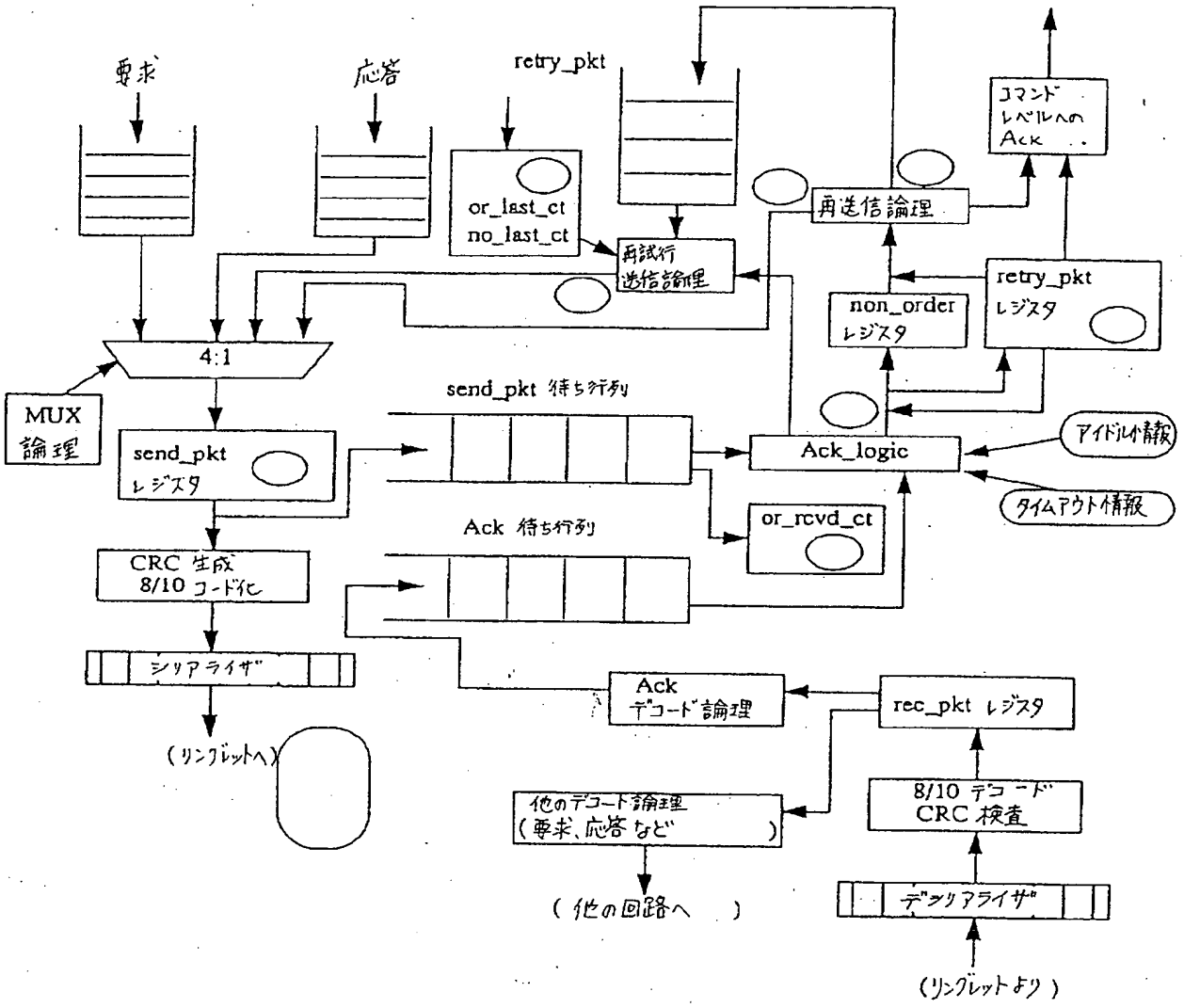
Seq 更新


ack_pkt = 1
Y
seq_reg.verif = 1
Y
seq_reg.sso = 1
Y
test_pkt_reg
.plabel.sso = 1
Y
test_pkt_reg
.plabel.seq =
(seq_reg.seq +1)
/*Add modulo 64*/
N
Y
updole 2
test_pkt_reg.plabel.seq
.GT. seq_reg.seq
SET:
seq_reg.verif = 1
seq_reg.sso = 1
seq_reg.seq =
test_pkt_reg.plabel.seq
seq_reg.bzseq =
test_pkt_reg.plabel.bzseq
seq_reg.busy = seq_reg.busy
seq_reg.ocpt = seq_reg.ocpt
rec_pkt_reg.producerId =
test_pkt_reg.producerId
seq_wr_vol = 1

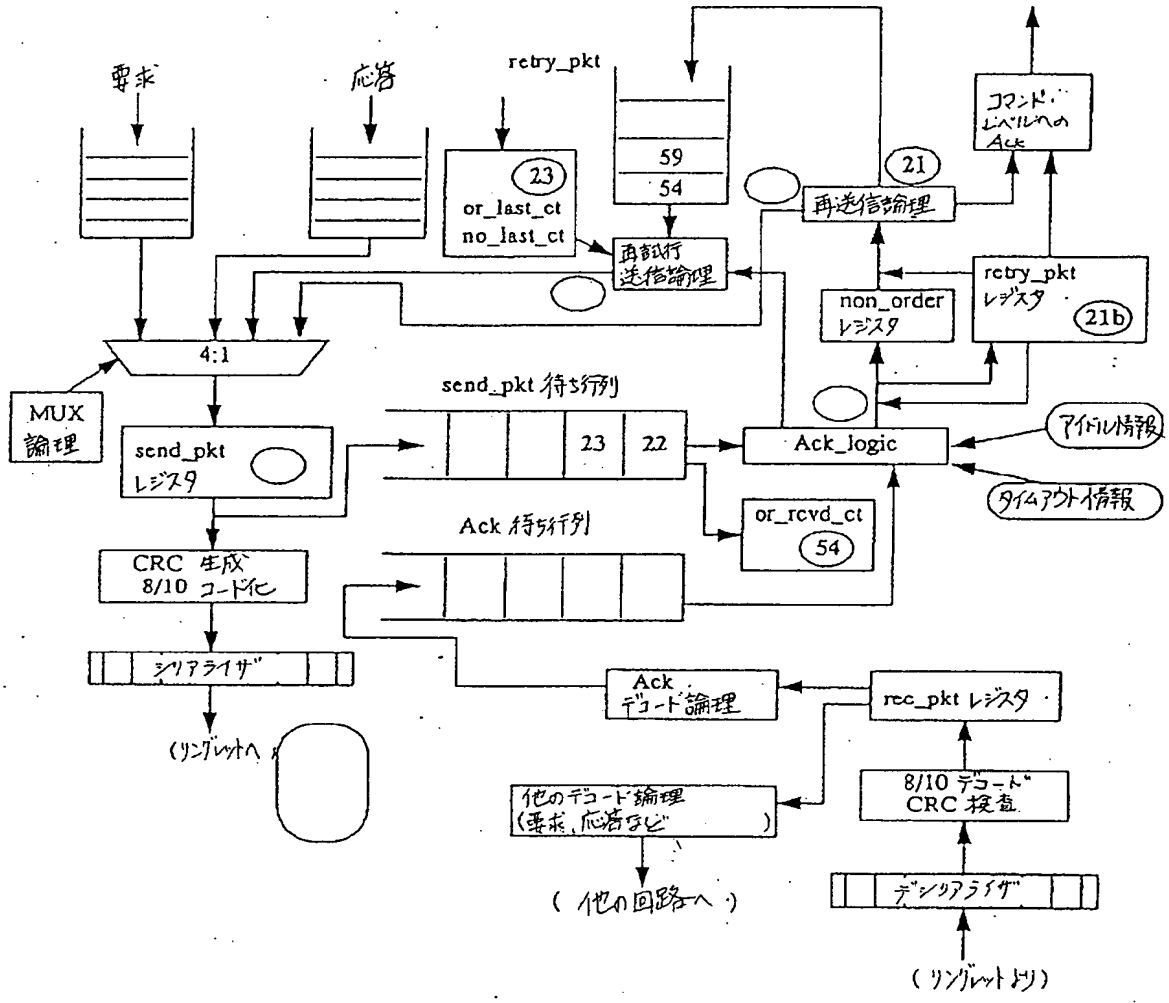
【図３９】


Seq 更新
update2
test_pkt_reg.pLabel
.seq .GT.
seq_reg.acpt
N
Y
test_pkt_reg
.pLabel.bzseq =
seq_reg.bzseq
Y
N
seq_reg.busy = 1
Y
SET:
seq_reg.verif = 1
seq_reg.sso = 1
seq_reg.seq =
test_pkt_reg.pLabel.seq
seq_reg.bzseq =
test_pkt_reg.pLabel.bzseq
seq_reg.busy = 0
seq_reg.acpt =
test_pkt_reg.pLabel.seq
rec_pkt_reg.producerId =
test_pkt_reg.producerId
seq_wr_val = 1
SET:
seq_reg.verif = 1
seq_reg.sso = 1
seq_reg.seq =
test_pkt_reg.pLabel.seq
seq_reg.bzserq =
test_pkt_reg.pLabel.bzseq
seq_reg.busy =
seq_reg.busy
seq_reg.acpt =
seq_reg.acpt
rec_pkt_reg.producerId =
test_pkt_reg.producerId
seq_wr_val = 1

【図４０】


要求
応答
retry_pkt
or_last_ct
no_last_ct
再試行
送信論理
再送信論理
コマンド
レベルへの
Ack
retry_pkt
レジスタ
non_order
レジスタ
4:1
MUX
論理
send_pkt 待ち行列
send_pkt
レジスタ
Ack_logic
アイドル情報
タイムアウト情報
or_rcvd_ct
Ack 待ち行列
CRC 生成
8/10 コード化
シリアライザ
(リングレットへ)
Ack
デコード論理
rec_pkt レジスタ
他のデコード論理
(要求、応答 など)
8/10 デコード
CRC 検査
デシリアライザ
(他の回路へ)
(リングレットより)

【図４１】



STATE

•or_last_ct: last ordered packet sent

•or_rcvd_ct: ordered packet received count, frozen if busy retry required (detected in resend logic)

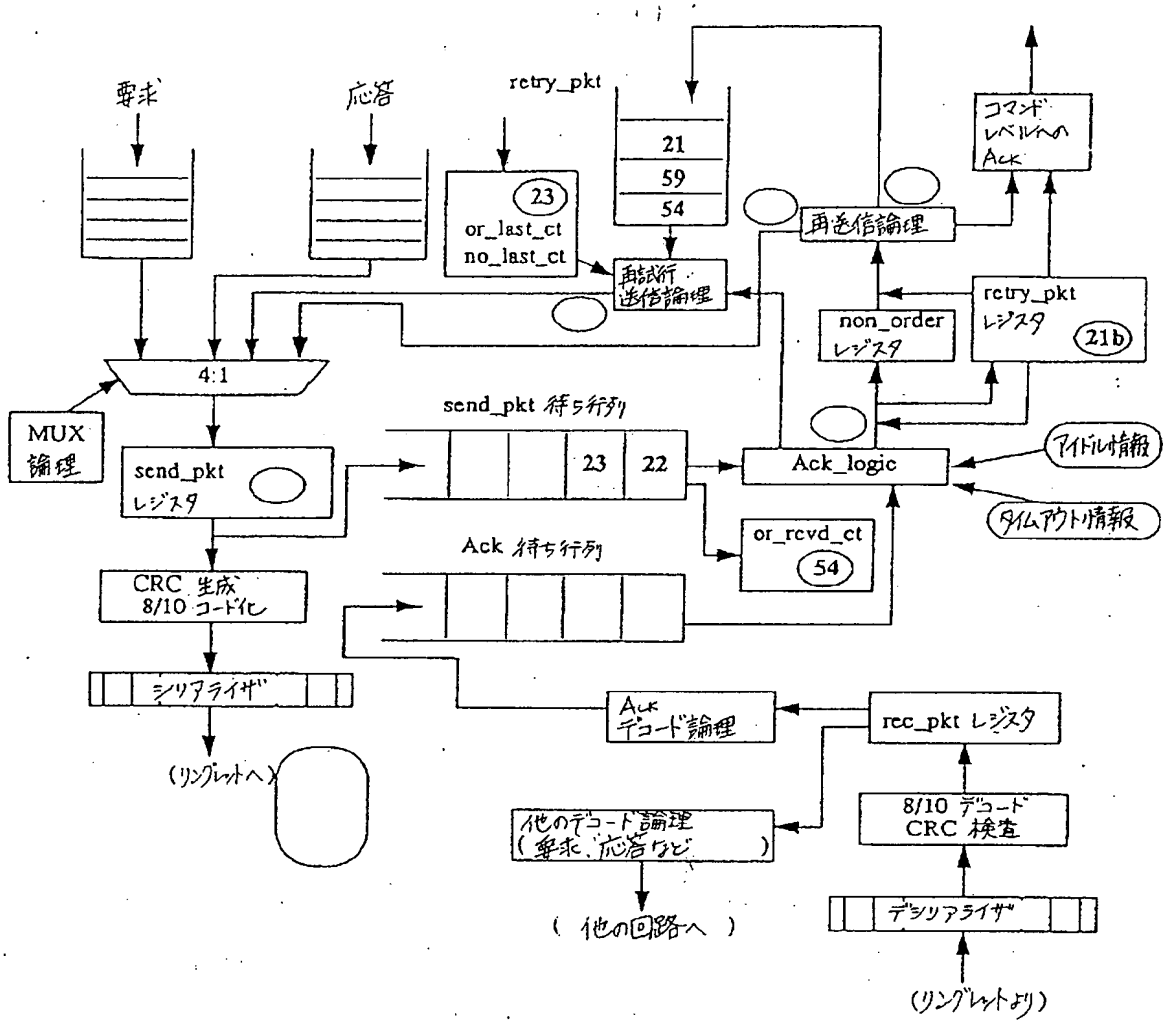
•Previous busy packets 54 and 59 (modulo 64) are in retry_packet queue.

Note: (or_last_ct - or_recd_ct) = (23-59) modulo 64
= 28, and 28 ≤ threshold for (or_last_ct - or_recd_ct) difference.

•Detect ack_busy in resend logic

•Add packet to retry_pkt queue

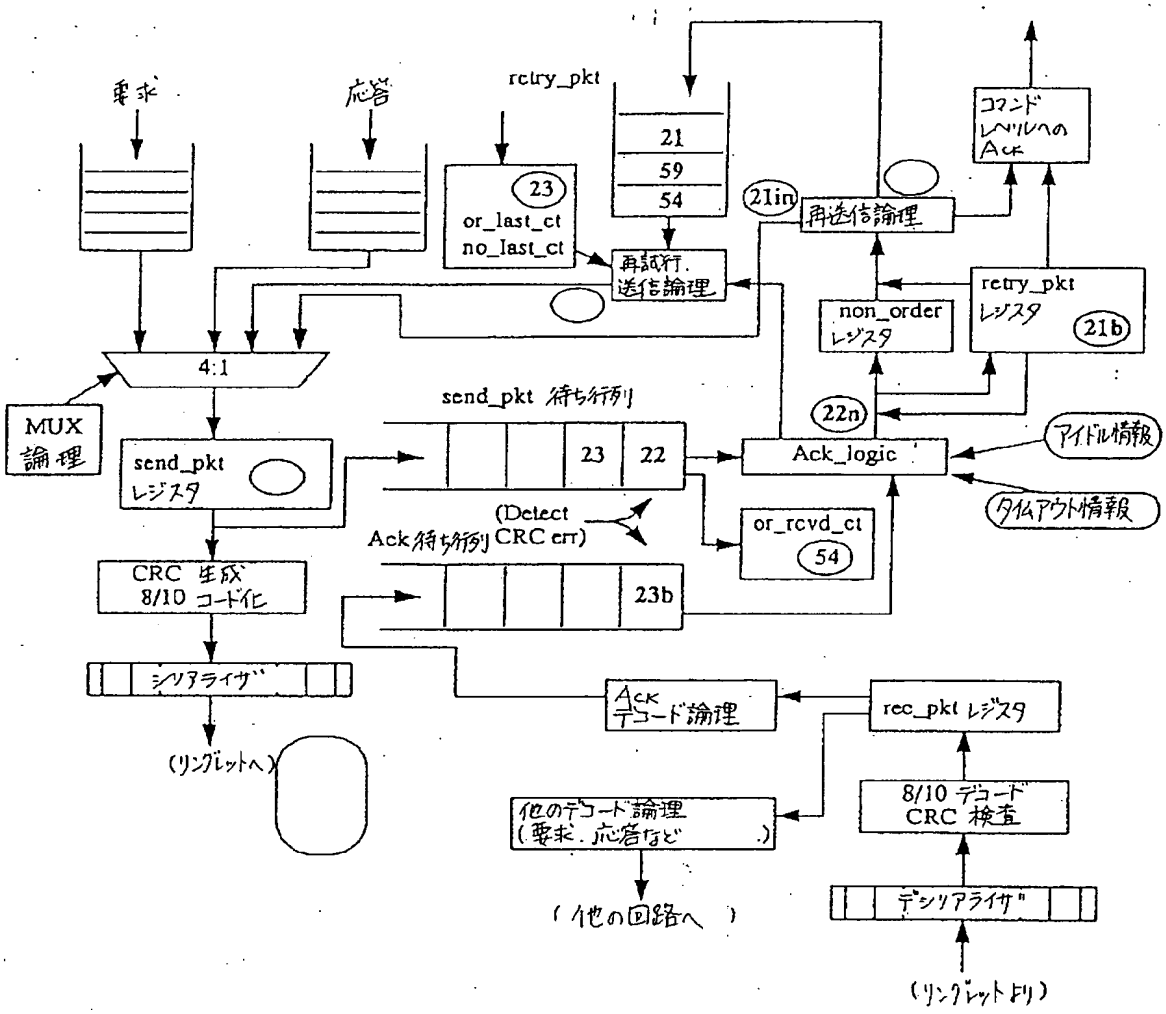
【図４２】



STATE

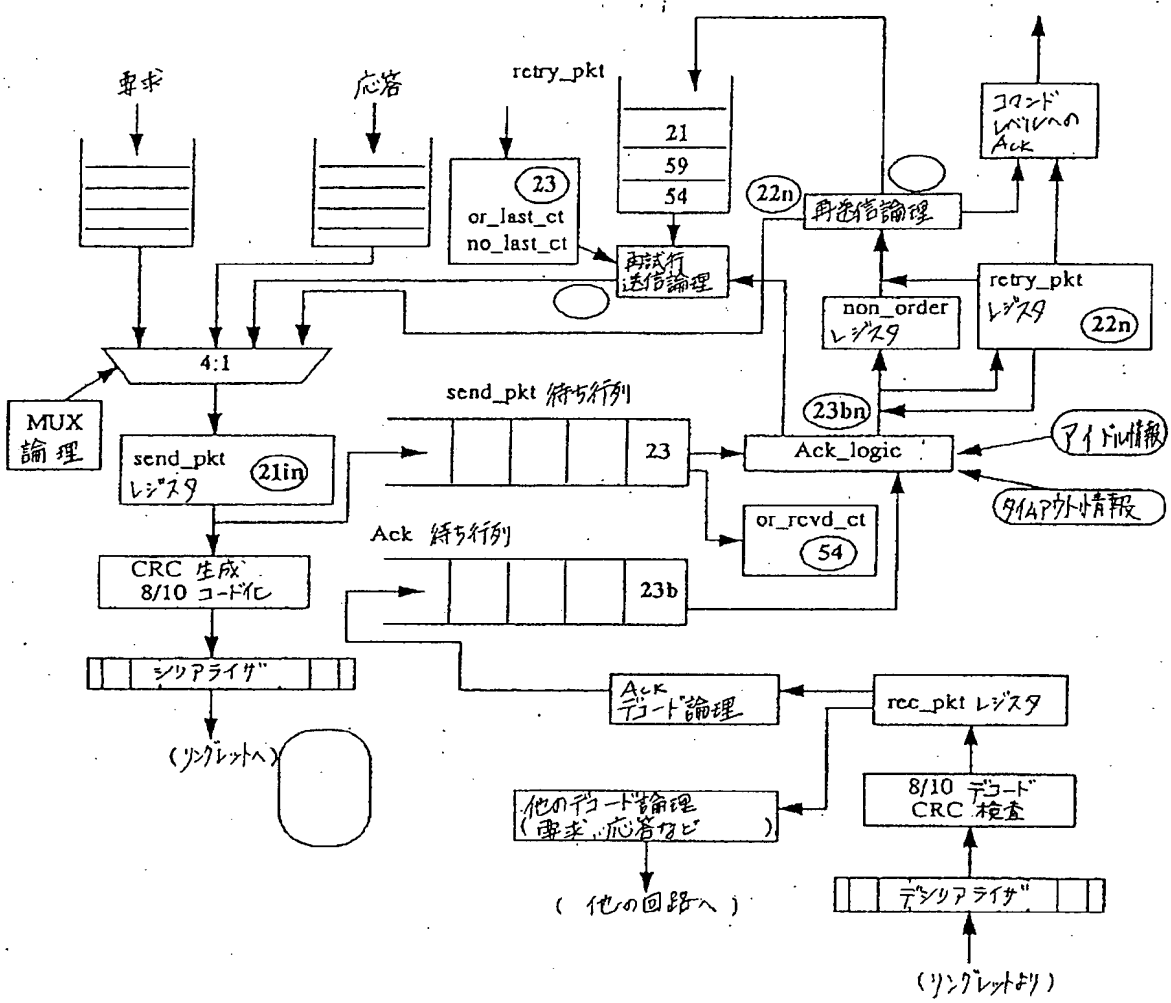
•Detect retry_pkt queue exceeds threshold. Set busy_loop.set$ and CRC_ack_chk.

【図４３】



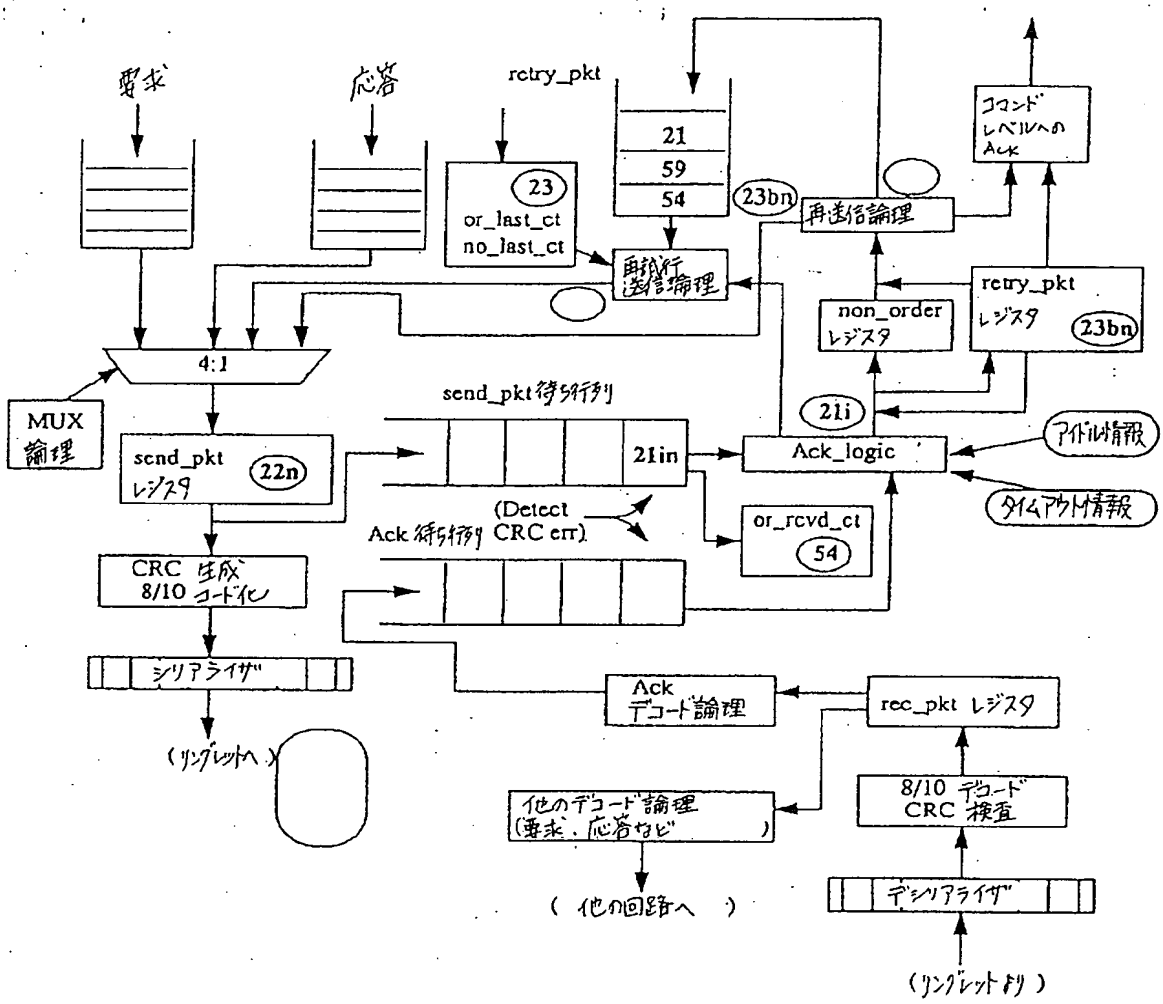
STATE
*Detect CRC error at head of send_pkt queue. Set CRC_err_retry control bit.
*CRC_ack_chk active.

【図４４】



STATE
*CRC_ack_chk active.
*Packet 21 inhibited from retransmission on the ringlet.
*CRC_err_retry control bit active.

【図４５】



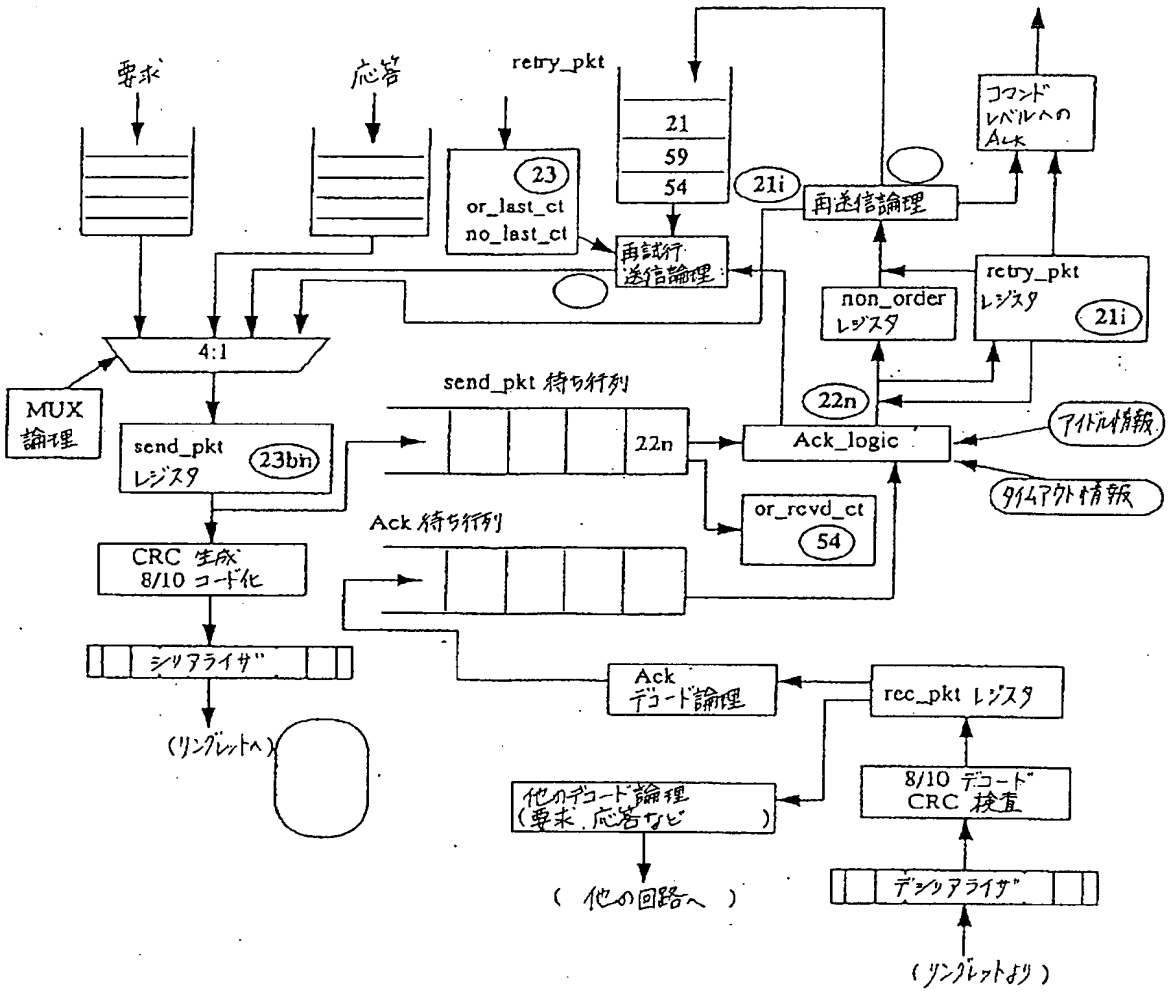
STATE

*CRC-ack-chk reset condition detected through "init_err" state at head of send_pkt queue

*CRC_init_err to be set because CRC_err_retry is active

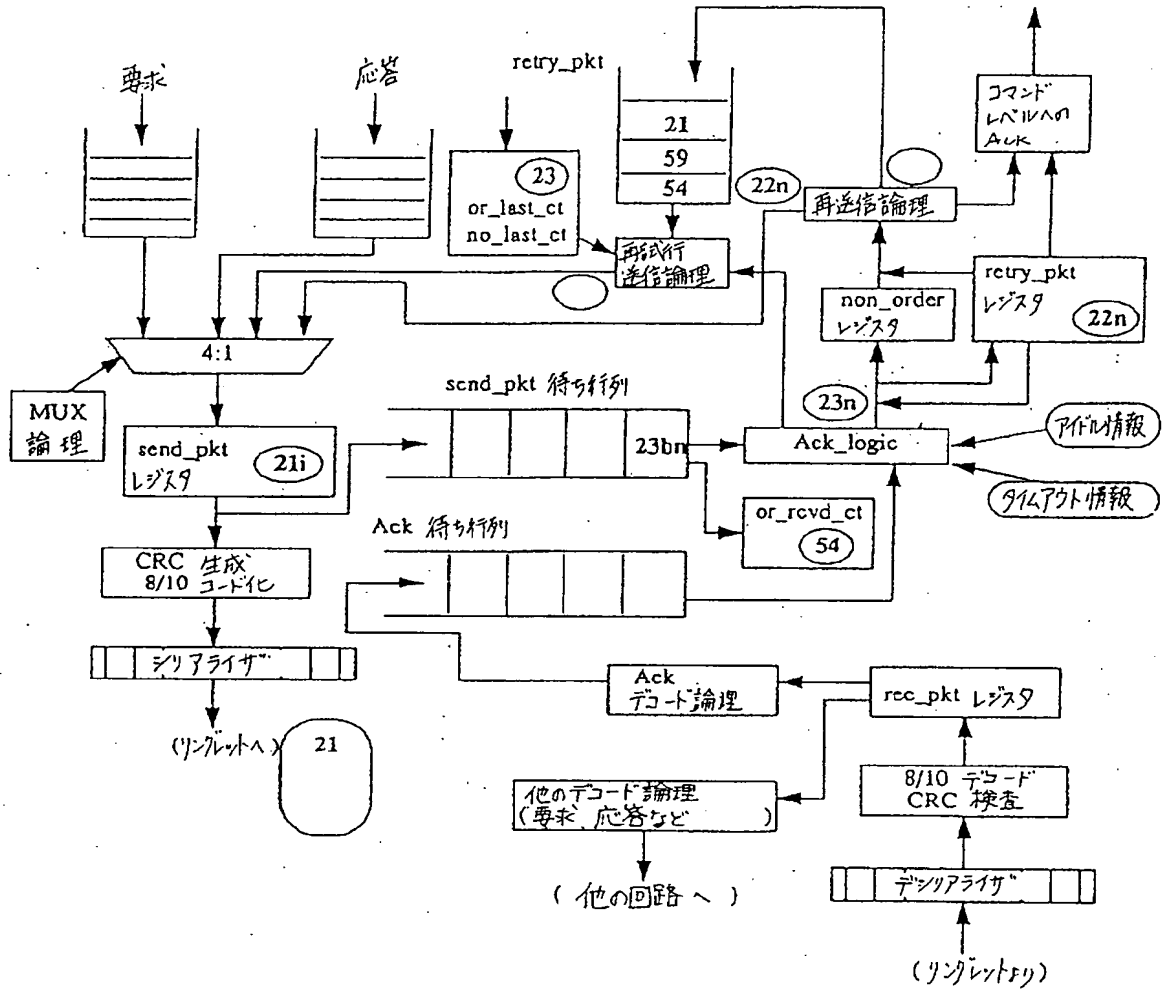
*CRC_err_retry to be reset

【図４６】



STATE
*CRC_init_err active

【図４７】



STATE

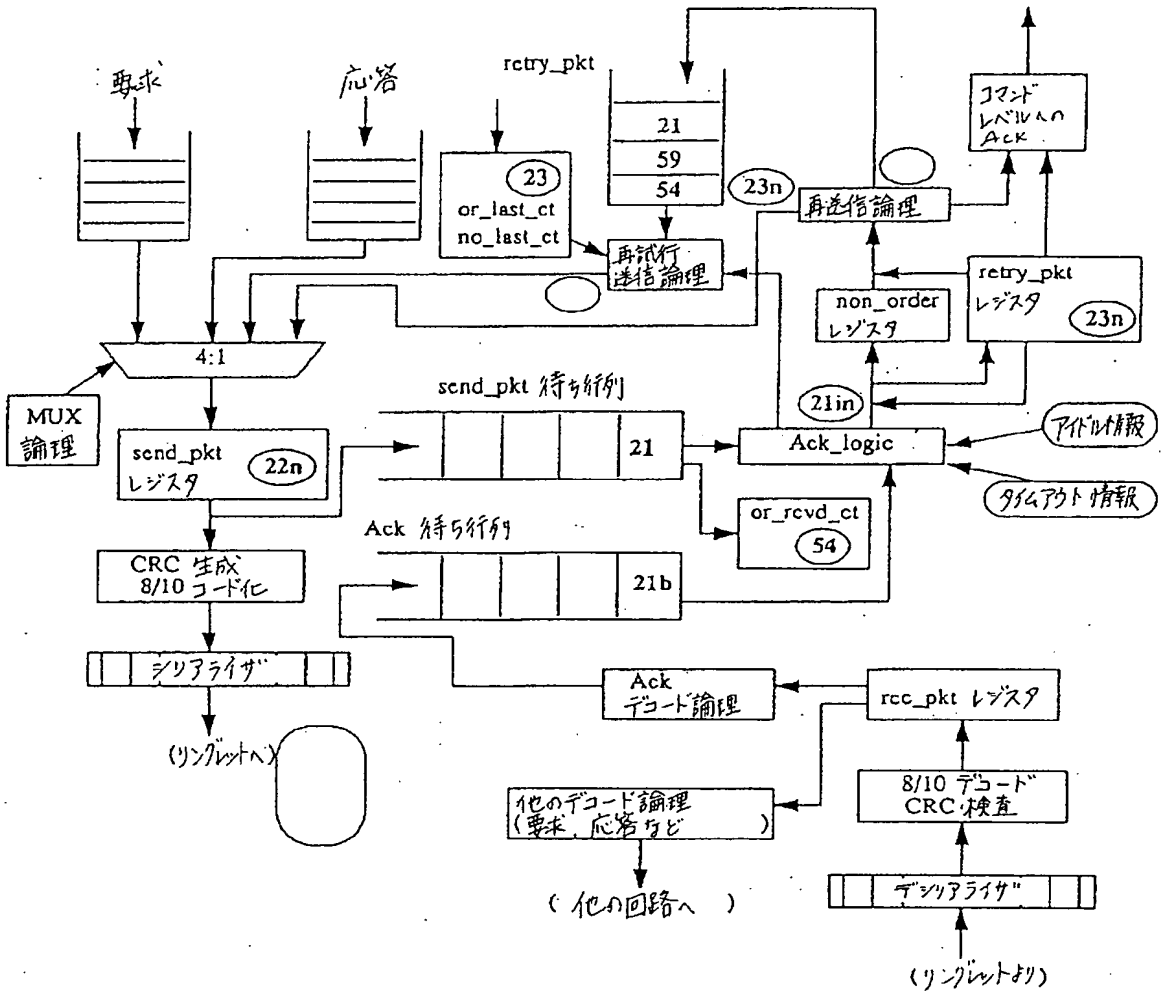
*Busy retry loop -- pending

*CRC_init_err active

--There is no need to maintain "busy" acknowledgment in this state, since acknowledgment on the error retry packet will be used to determin "done" or "busy" packet disposition.

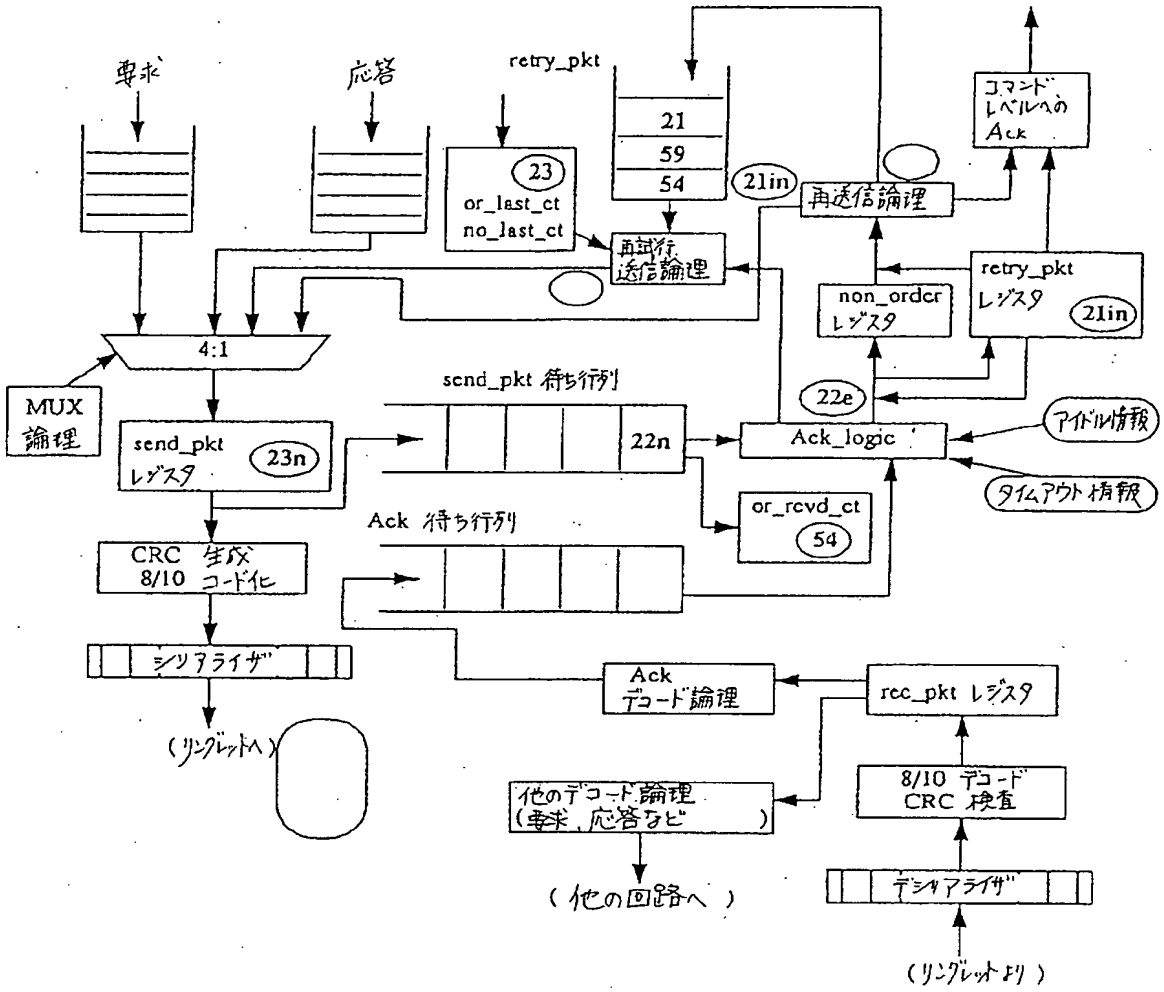
--"Known good packet" 21 is transmitted to the ringlet and enqueued in send_pkt queue to test acknowledgment.

【図４８】



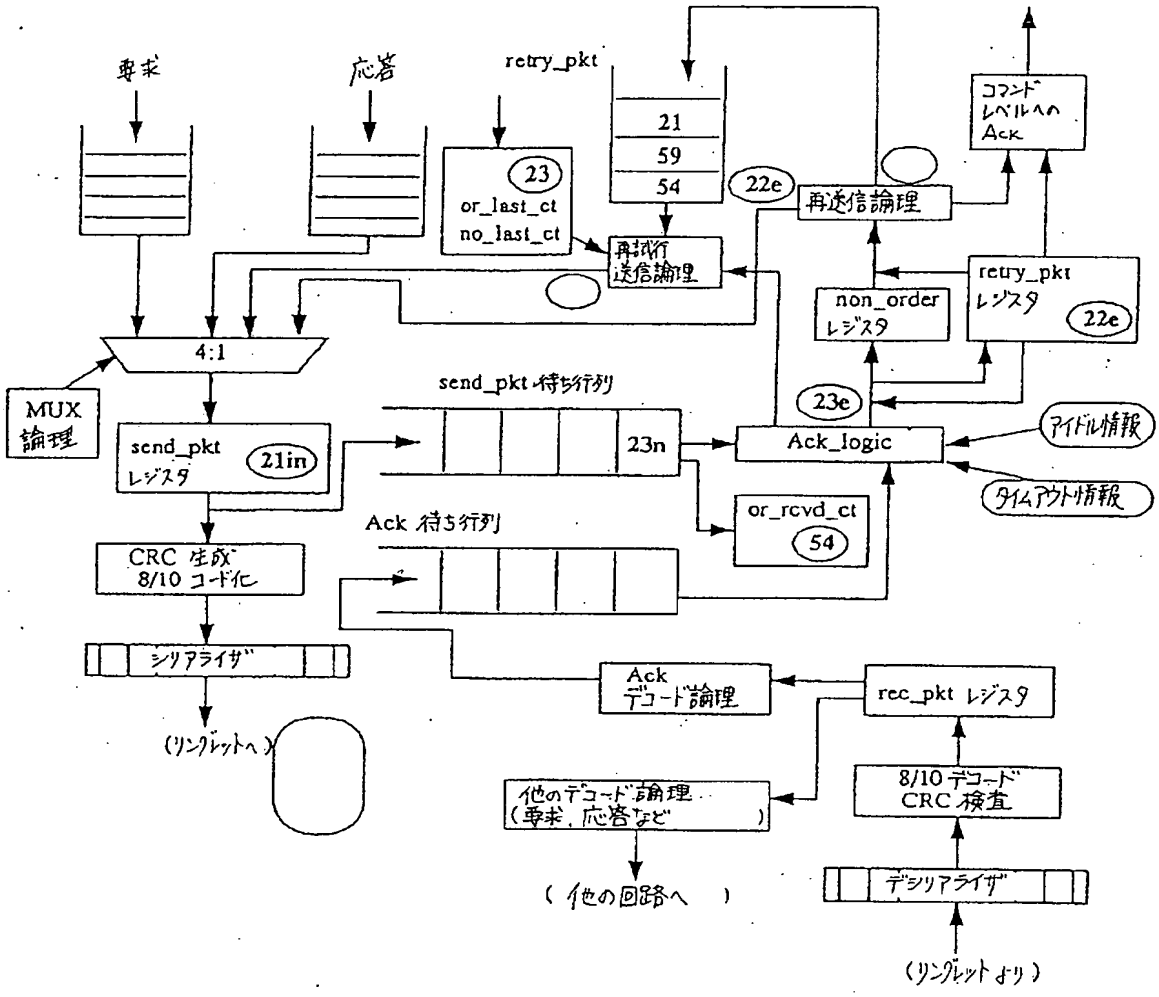
STATE
*Busy retry loop -- pending
*CRC_init_err is reset by detecting valid ack for "known good" packet 21.
*CRC_err is set.

【図４９】



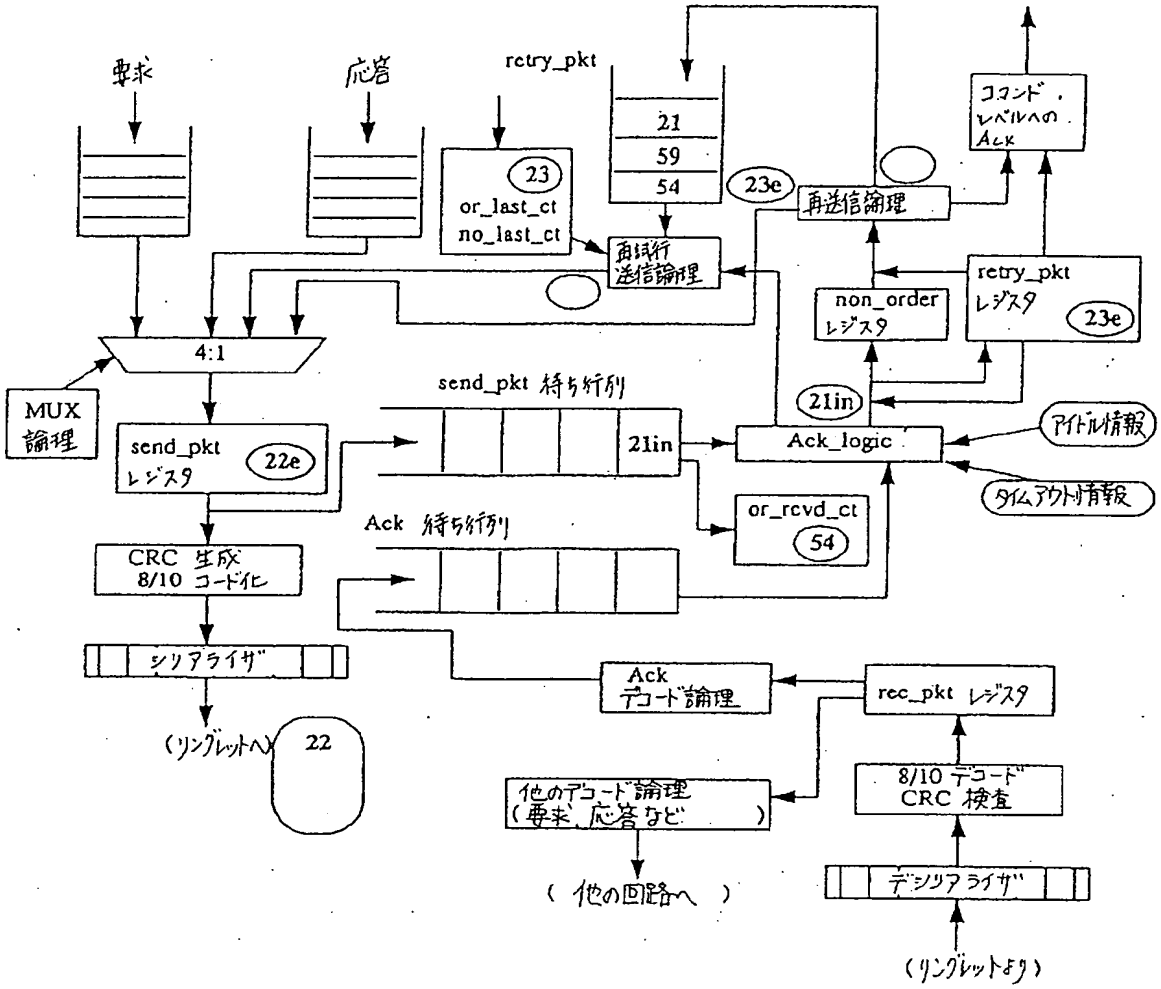
STATE
*Busy retry loop -- pending
*CRC_err active

【図５０】



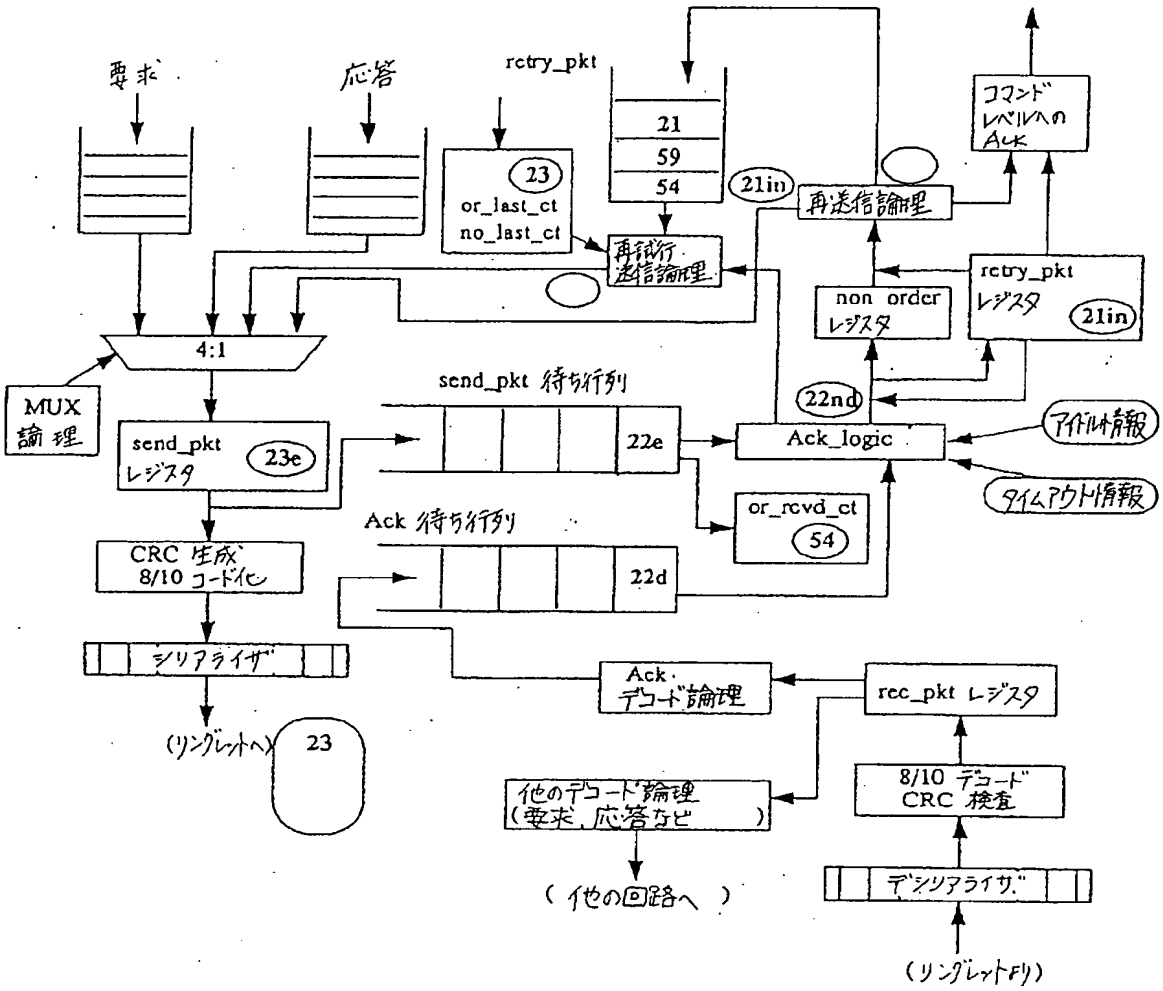
STATE
*Busy retry loop -- pending
*CRC_err active
--Packet 21 is cycled through but is not resent (no xmit state)

【図５１】



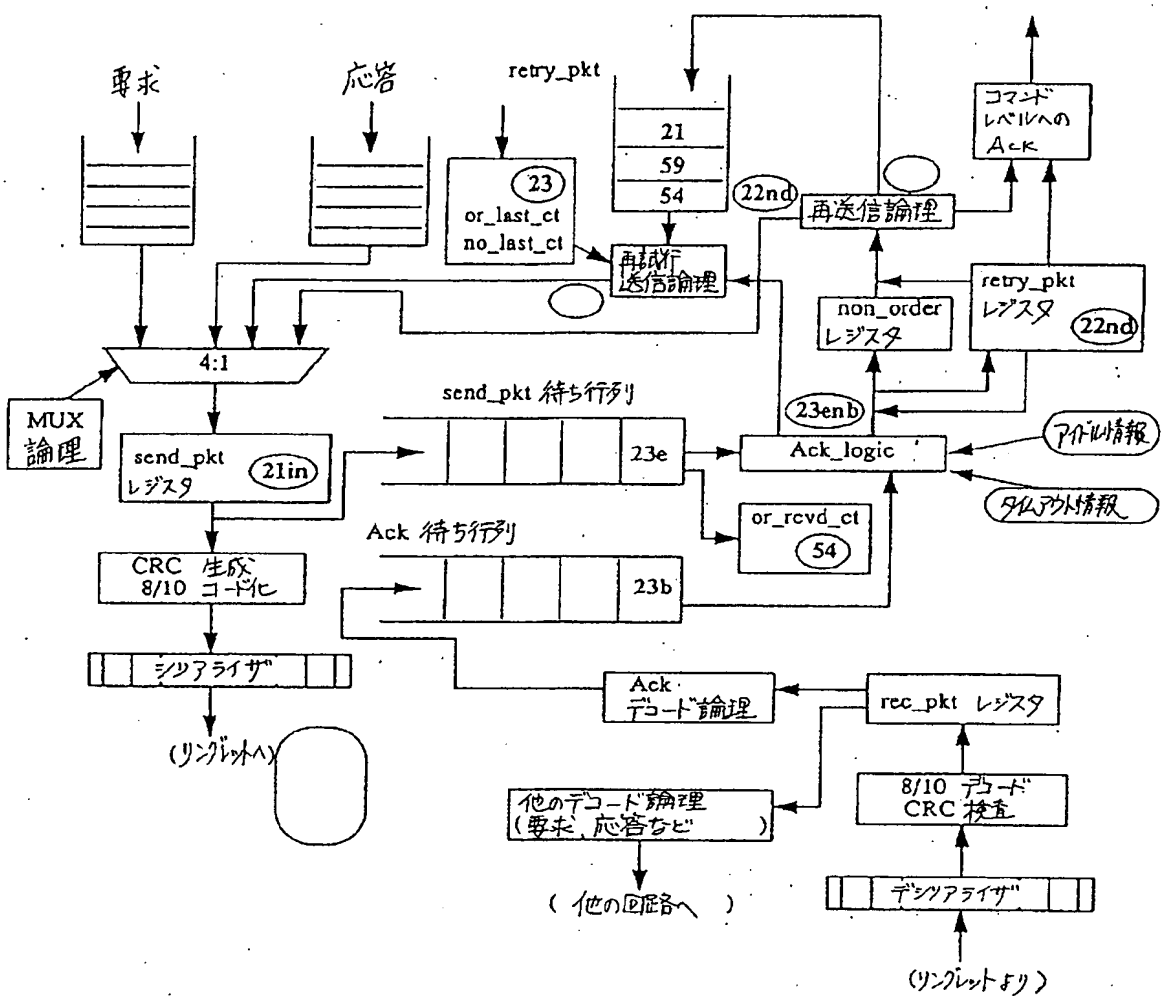
STATE
*Busy retry loop -- pending
*CRC_err is reset by detecting (in Ack_logic) "known good" packet 21 with ("init_err" and "no_xmit") state.
*CRC_ack_chk is started
*Packet 22 is retransmitted

【図５２】



STATE
*Busy retry loop -- pending
*CRC_ack_chk is active
--Retried packet 22 gets "ack_done". Remaining "acks" in retry loop must be checked before 22 can be retired. "ack_done" state is saved.

【図５３】



STATE

*Busy retry loop -- pending

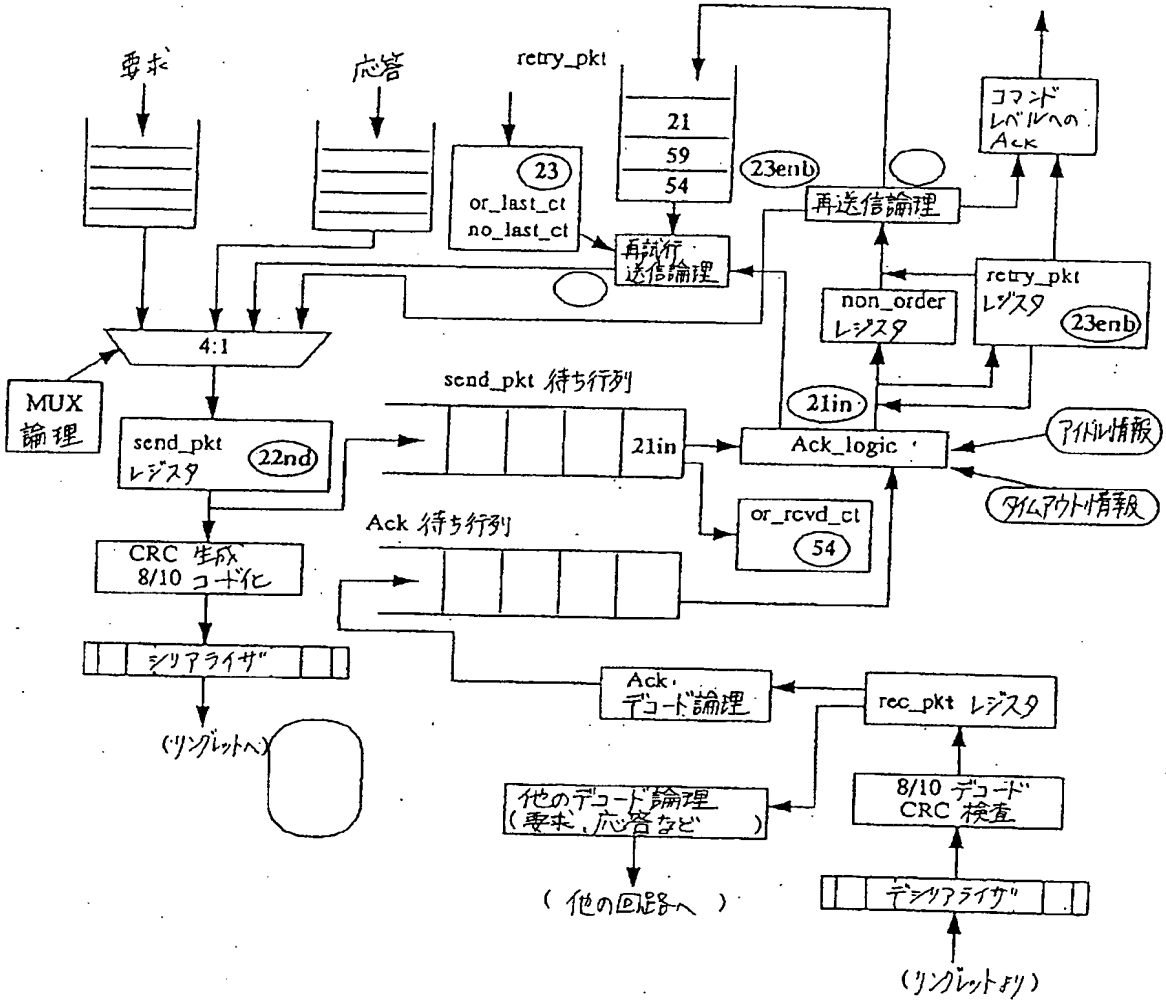
*CRC_ack_chk is active

--Retried packet 23 gets "ack_busy" again.

--Packet 21 is not retransmitted.

--Packet 23, the head of the send_pkt queue, is recognized as the last packet by comparing its seq field with the *(or_last_ct) register. Its state is set to (err_ret and no_xmit). Its busy acknowledgment is maintained with the "busy" state.

【図５４】



STATE

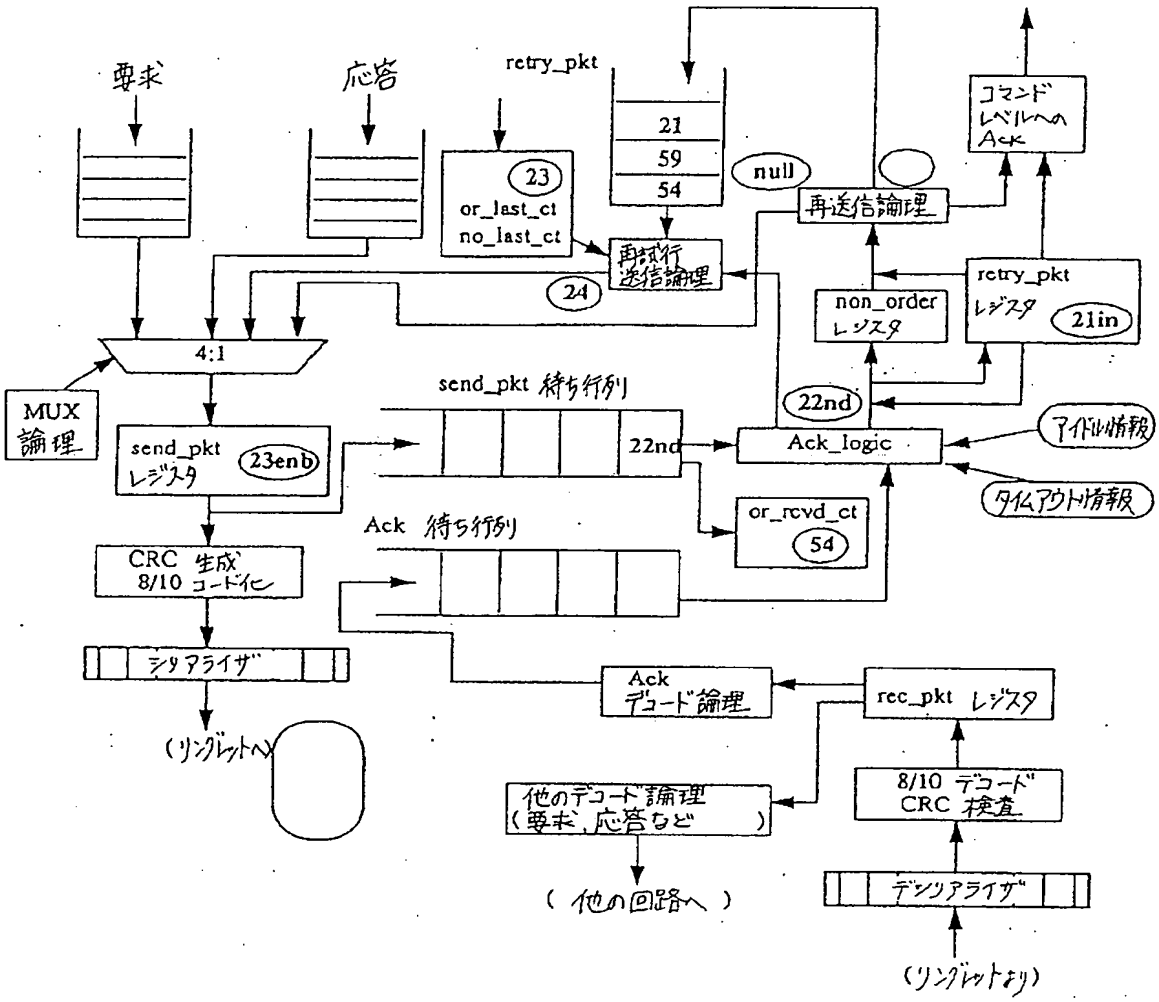
*Busy retry loop -- pending

*Reset CRC_ack_chk and busy_loop_pnd by detecting "known good" packet 21 with (init_err and no_xmit) state and no error detected during CRC_ack_chk loop.

--Retried packet 23 gets "ack_busy" again.

--Set CRC_err_end, residing in resend logic, to detect last packet when it cycles through retry_pkt_reg.

【図５５】



STATE

*Busy retry loop starts

--Retry_send logic modifies seq field for head of retry_pkt to be (or_last_ct + 1)

--Increment (or_last_ct) by 1..

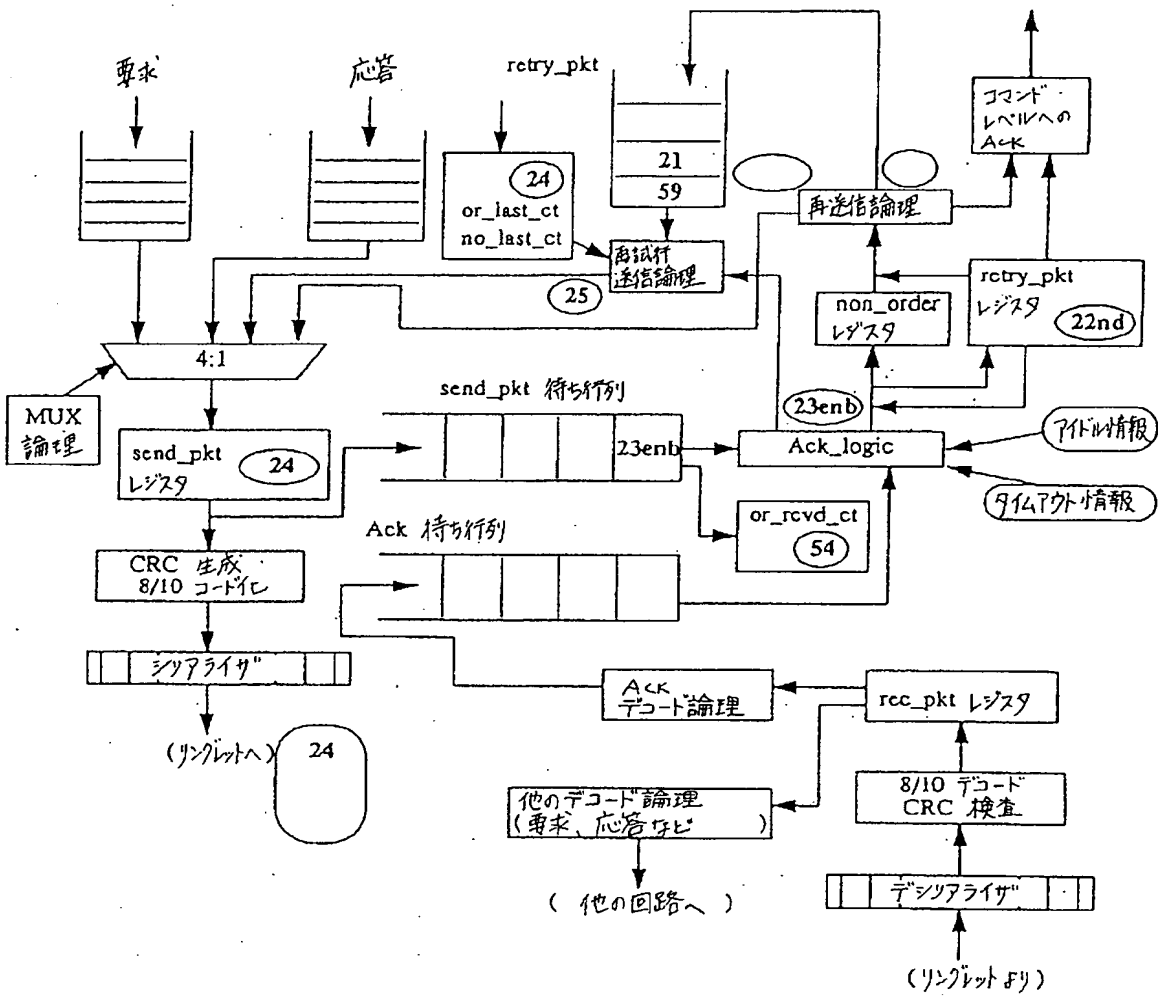
--Dequeue packet 54 from retry_pkt queue.

*CRC_err_end active in resend logic

--Drop "known good" packet

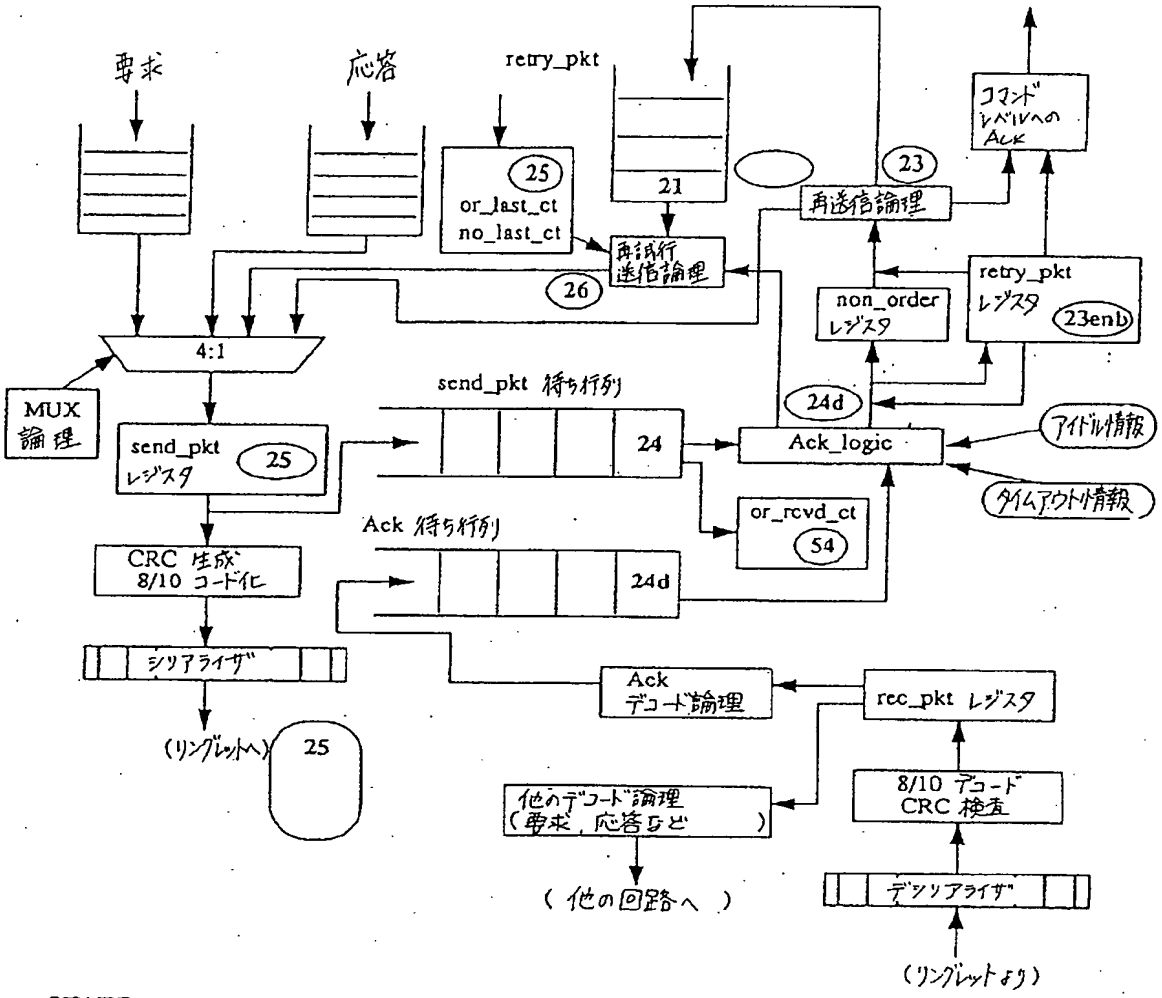
*Packet 24 assigned new "bz seq" field as first retried busy product

【図５６】



STATE
*Busy retry loop active
*CRC_err_end active in resend logic
--Packet 22 completed
--Packet 24 transmitted to ringlet
*Busy retry packet 59 assigned sequence number 25

【図５７】



STATE

*Busy retry loop active

--Packet 24 "done" acknowledge detected

*CRC_err_end loop

--Packet 23 detected as last packet with "last packet" state (err_rst and no_xmit) and (busy or done).

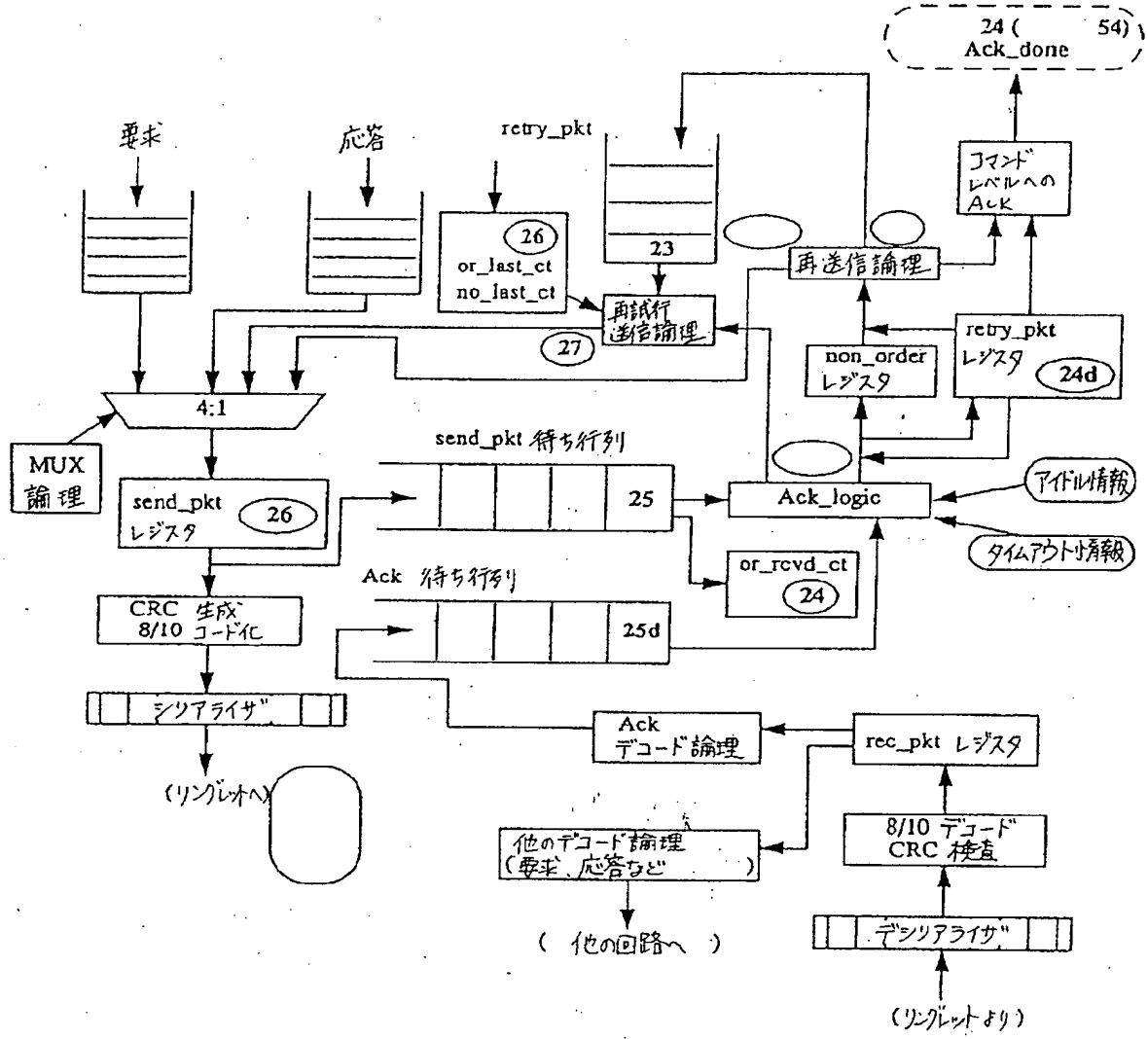
--"Last packet" state resets CRC_err_end loop

--"Last packet" state sets "retry_pkt_cmplt" control bit for busy retry loop. When "retry_pkt_cmplt" is set and the retry_pkt queue is empty, busy retry loop is reset.

*Retried packet 25'sent to ringlet.

*Busy retry packet 21 assigned sequence number 26

【図５８】



STATE

*Busy retry loop active with "retry_pkt_cmplt"

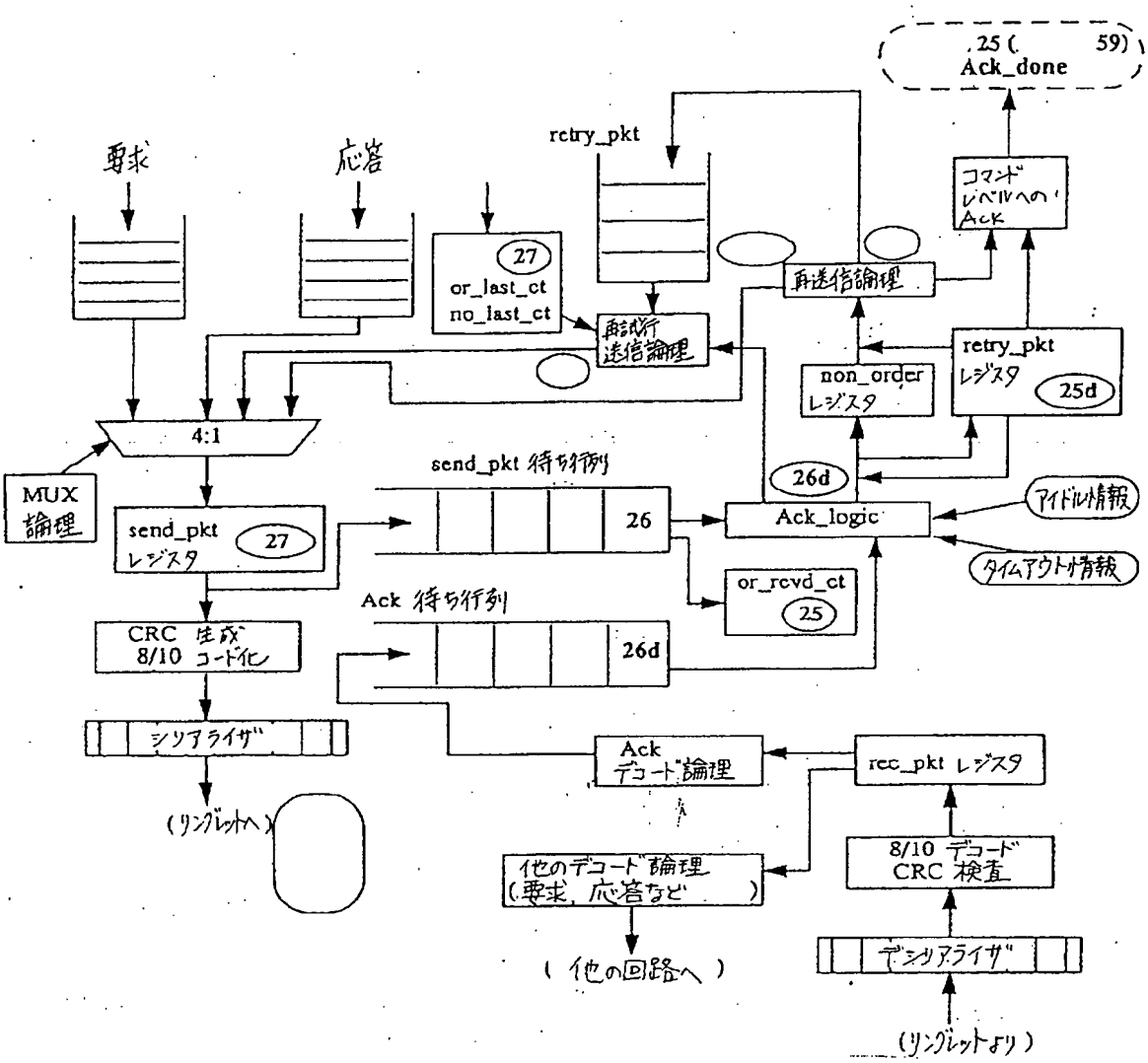
--Packet 23 in head of retry_pkt queue assigned new seq field 25 from (or_last_ct) + 1

--(or_last_ct) incremented by 1

--Packet 23 dequeued

*Packet 24 retried as ack_done

【図５９】

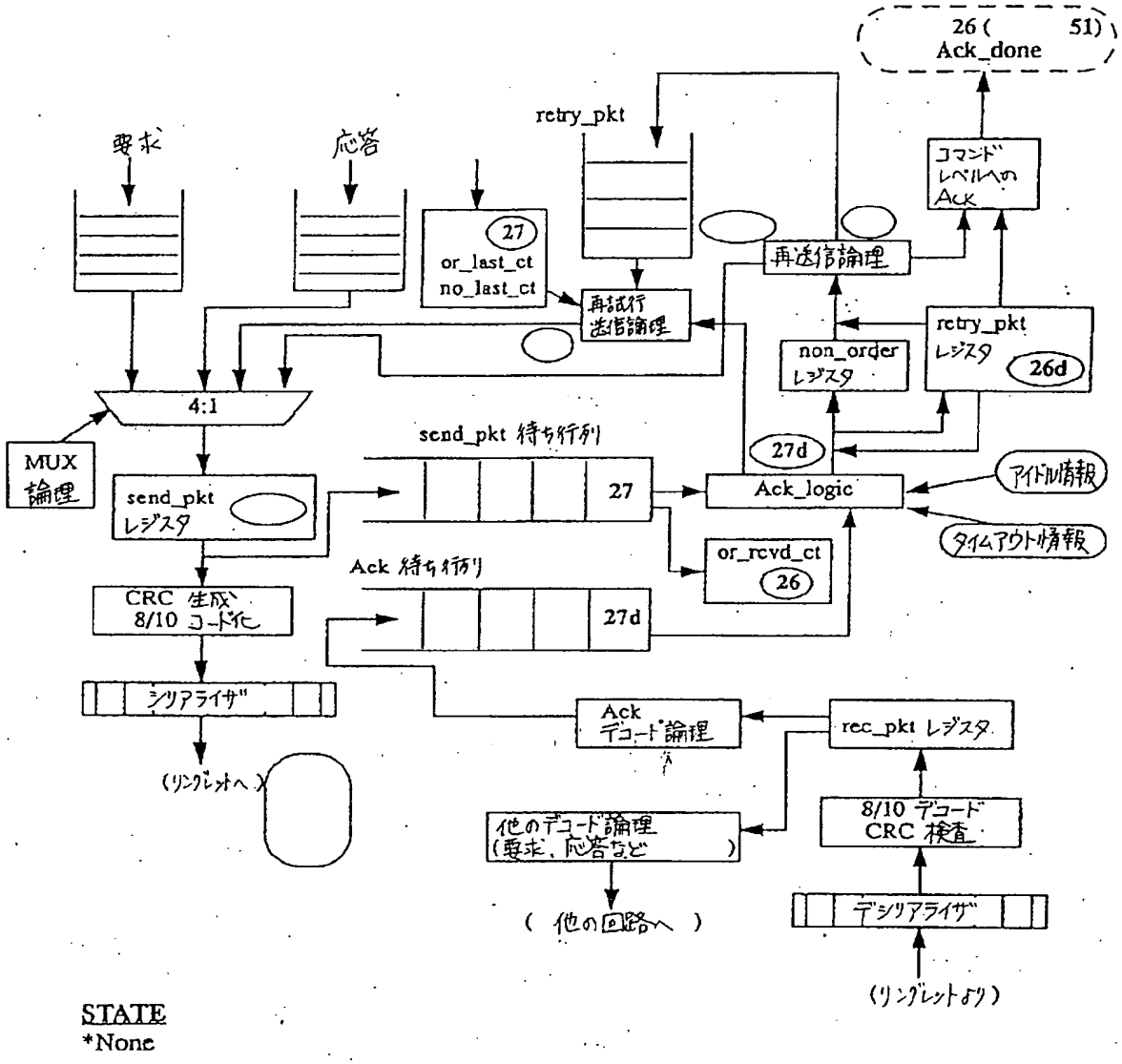


STATE

*Busy retry loop active with "retry_pkt_cmplt"

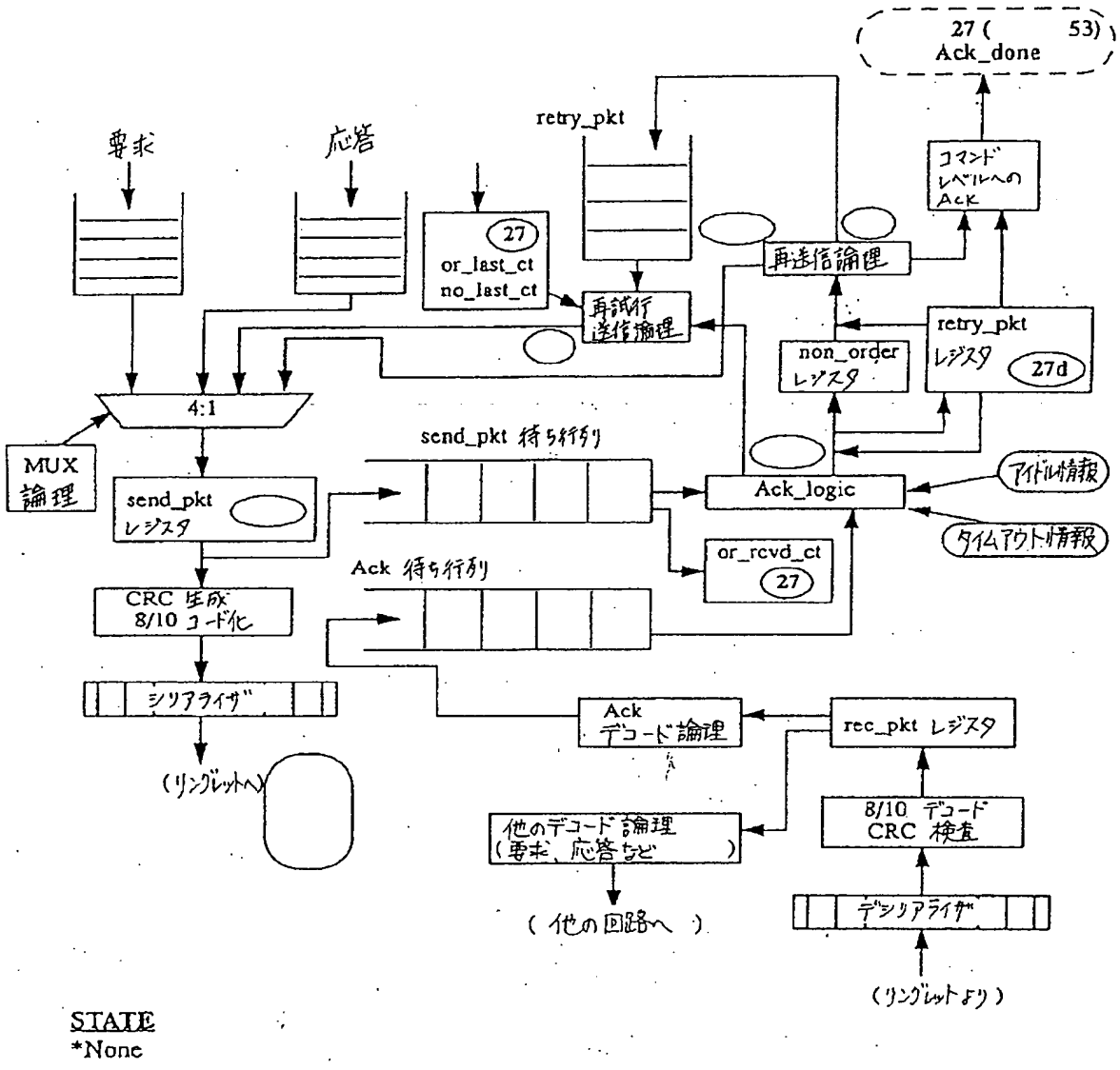
--Empty retry_pkt queue detected with "retry_pkt_cmplt" - resets busy retry loop and "retry_pkt_cmplt"

【図６０】



STATE
*None

【図６１】



STATE
*None

フロントページの続き

(71)出願人 591064003
901 SAN ANTONIO ROAD
PALO ALTO, CA 94303, U.
S. A.

【外国語明細書】

System for Maintaining Strongly Sequentially Ordered Packet Flow in a Ring Network System With Busy and Failed Nodes

Background of the Invention

The present invention is directed to a system for support of packet transmission in processor-based networks using serial interconnects. In particular, the system of the invention provides dynamic ordering support for packets in response to errors in such networks, in particular in ringlet topologies, accommodating both busy conditions at nodes on the network and nodes that fail to respond or respond for long periods of time with busy acknowledgments, due either to overload or node failure.

Serial interconnects in computer systems are subject to a number of different types of service interruptions. For example, when a node on a network encounters a CRC (cyclical redundantcy check) error in a packet, that packet cannot be accepted. The node that sent the packet learns, generally indirectly (such as by a timeout or through the use of idle packets) of the error, and eventually must resend the packet.

Resending the packet may not be a simple matter, especially if the network implements an ordering scheme, such as relaxed memory ordering (RMO), strong sequential ordering (SSO), or orderings of other or intermediate stringency. In a packet-switched network with such an ordering scheme, packets preceding and following a packet giving rise to a CRC error may need to be resent by the producer node to the target node.

A particular problem arises when such a packet contains a nonidempotent command, i.e. a command which, once it is executed at the target node, changes the state of that node, such that reexecution of the command at that node would yield different results from the first execution; in this case, if the command is resent to the node and executed again, undesired or unforeseen results are likely to take place.

Thus, a system is needed wherein errors in packets can be accommodated by resending the packets to the target node, while maintaining support for idempotent commands. In particular, such a system is needed that also supports various levels of ordering schemes.

A particular need is present for such a system that can accommodate ill-behaved nodes,

i.e. nodes that fail or take unacceptably long to reply. In a topology where a single receiver is poorly behaved, this node can impede forward progress of the new packets by forcing the repeated sending of the same busy packet to complete before new packets are introduced. Thus, a sysetm is needed that deals with this potential pitfall.

Summary of the Invention

A system is presented which provides for resending of packets that have resulted in CRC errors at the same time as dealing with busy acks, by maintaining a state at each receive node of all known good packets. When packets need to be resent, the local nodes know whether they have processed the resent packets before, and know which the last known good packet was, and in this way are able to avoid reprocessing already executed commands, including nonidempotent commands. A busy loop can be effectively suspended while the error loop is executed, and once a reordering of the packets takes place, the busy loop can be completed. In addition, the system provides for an ill-behaved, e.g. nonresponding node, which does not send any response back to the producer node. Such an ill-behaved node can be effectively neutralized by a queued retry policy, wherein busy packets are accumulated in a retry packet queue, until some predetermined threshold is reached, after which the continually busy node is effectively removed from the system by the producer node ending its attempts to retry the packets to that node. In this way, the system can proceed with other outstanding packets.

This application thus relates to a complex system achieving the resolution of three simultaneously occurring problems on a ringlet network: error conditions, busy conditions at one or more nodes, and the failure or overloading of a node. A suitable error retry mechanism is described both herein and in applicant's copending patent application filed July 1, 1996, entitled System for Dynamic Ordering Support in a Ringlet Serial Interconnect by van Loo et al. A system based upon such an error retry mechanism and at the same time able to handle busy retry operations is described in applicant's copending patent application filed July 1, 1996, entitled System for Preserving Sequential Ordering and Supporting Idempotent Commands in a Ring Network with Busy Nodes by van Loo et al. Each of these patent applications is incorporated herein by reference.

Brief Description of the Drawings

Figure 1 is a block diagram showing ringlets on a larger network.

Figure 2 is a block diagram showing a single ringlet, illustrating commands sent and received on the ringlet.

Figures 3-4 illustrate commands sent and received in the cases of strong sequential ordering (SSO) and SSO with nonidempotent requests or commands, respectively.

Figure 5 shows the ringlet of Figure 2, showing the data structure of send packets and acknowledgments.

Figure 6 shows the ringlet of Figure 2, illustrating the resending of packets.

Figure 7 shows the ringlet of Figure 2, illustrating the actions of a consumer node.

Figure 8 is block diagram illustrating the operation of the invention from the point of view of a send interface.

Figure 8A is a timing diagram illustrating a busy loop operation of the invention, with respect to a send port.

Figures 9-25 are logic diagrams illustrating logic subsystems of the system of the invention, and their operation, in which (in the informal drawings) hexagons are decision boxes and rectangles are steps at which variable values are set.

Figure 26 is a block diagram illustrating the operation of the invention from the point of view of a receive interface.

Figure 27A is a timing diagram illustrating receive packet timing.

Figure 27B is a timing diagram illustrating acknowledge timing.

Figures 28-38 are logic diagrams illustrating logic subsystems of the system of the invention, and their operation, with respect to receive nodes.

Figure 39 is a block diagram indicating both the structure and the flow of packets in a preferred embodiment of the invention.

Figures 40-60 illustrate successive states of the block diagram of Figure 39 according to the operation of the system of the invention.

Description of the Preferred Embodiments

Figure 1 shows a standard ringlet 1 with four nodes A-D, each with transmit and receive capability. Requests and responses are sent out by a producer node, e.g. node A, while acknowl-

edges (e.g. ack_done and ack_busy) are sent by a consumer node, e.g. node C. Node C may also communicate with other ringlets 2, 3 and 4, in a conventional fashion. The present invention is directed in particular to solutions for packets that are subject to errors in a ring network such as ringlet 1, which can cause unpredictable packet delivery and acceptance, since, although a producer knows what it has sent, it does not know (except by indirect means), whether a packet for which no acknowledgment has been received was in fact processed by the intended node.

When ordering is required, the challenge is greater. Figure 2 shows commands sent in a particular order A, B, C... by node A, and because of ringlet and transmission vagaries, reception in a different order, e.g. B, A, A, A,.... A well-behaved system should be able to reorder the sent packets and execute them properly, in the originally sent sequence.

Figure 3 illustrates strong sequential ordering (SSO): no packet is received out of the order in which it was sent -- although it may appear twice, it does not appear out of order. Figure 4 illustrates the use of both SSO and nonidempotent commands; not only is the sequence preserved, but because of the nature of nonidempotent commands, no command in a conventional system may be executed twice by the destination node. Note in Figure 4 that {producerId, pLabel} are common to send_pkt and acknowledge; also, {producerId, pLabel} are visible to each node, absent errors.

Figure 5 shows date structure suitable for applicant's solution to the above challenges; the packets include fields called producerID and pLabel, whose details are discussed below. These two fields are included in the acknowledge packets sent back from the consumer to the producer.

The present invention resends packets, both known good (already accepted/done) packets and others, in a pattern as shown in Figure 6. Figure 6A summarizes at a high level the basic approach of the system of the invention: a known good SSO packet is resent, repeatedly if necessary, until a valid acknowledgment (which may be "done" or "busy") is received at the producer node. Then the producer node resends all other packets after the known good packet, even those that have been properly processed, i.e. for which ack_dones have been received. (See Figure 7.) As the acks come back from the resent packets, the producer node checks to see that all are valid acks -- i.e. "done" or "busy". If they are not all valid acks, then the resending process is repeated, until all acks are in fact valid. This ensures that all erroneous packets have been properly processed, and the system may then proceed with succeeding packets. Complicated challenges are presented in this approach to preserve SSO and support nonidempotent commands, and the solu-

tions are presented by the logic shown in Figures 8A-38, whose operation is illustrated in the state diagrams of Figures 39 et seq.

The present invention is enabled by the fact that the consumer node continually maintains the state for packet actions (accept and reject states) and for acks (done and busy); this goes for all packets and acks that each node sees. Thus, each consumer node is aware of the disposition for every packet traveling the ringlet at the time that each packet passed through that node. This information is important in the resend process. Moreover, each consumer node maintains a record of the sequence of packets that have gone through, and a sequence number of the last known good packet (for which an ack_done or ack_busy was issued). This provides a point to which the system can back up, if needed, when e.g. a CRC error is encountered. This will be more clearly seen in the discussion below of the example of Figures 39 et seq.

Figure 8 is a block diagram showing a suitable hardware environment for the present invention, and serves also to show the possible flow of packets in the producer node. It can be understood in detail in light of the flow/state diagrams of Figures 39 et seq. and the logic diagrams of Figures 8A-38.

In Figure 39, the fundamental blocks of the system are shown. In the absence of the SSO ordering extensions, this system would include all blocks with the following deletions and changes:

- Delete retry_pkt register, 7-190.
- Delete retry_pkt queue, 7-30.
- Delete or_last_ct counter inside 7-100; retain no_last_ct.
- Modify 4:1 mux to be 3:1 mux, 7-40.
- Delete or_rcvd_ct counter, 7-250.

Other blocks remain fundamental to the operation of a send node with no SSO support.

The basic operation of the system is as follows, with reference to both Figure 8 and the more detailed block diagram of Figure 39. Packets are inserted from the transaction layer into the link layer through the request and response FIFO queues. Packets are selected through the multiplexer (MUX) for (possible) transmission to the ringlet through the send_pkt register, the CRC generator and 8/20 encoder, and the serializer, to the ringlet, as illustrated.

Acknowledges for request and response send packets from this node, as well as incoming

request and response packets originating from other nodes but directed to this node, are received in the deserializer. After an 8/10 decoding and CRC check, incoming receive and acknowledge packets are registered in the rec_pkt register. Idle symbols are stripped before the rec_pkt register. CRC errors or 8/10 encoding errors invalidate any incoming receive and acknowledge packets. Acknowledges with CRC errors are not visible beyond the rec_pkt register, and received send packets from other nodes are processed so that they are not visible to the transaction layer. (See the receive buffer description.) The ack decode logic, detects acks addressed to this node for the send packets it had transmitted through its serializer.

Acks are queued in the ack queue, and matched with send packets in the send_pkt queue,, in the ack_logic block. In the absence of SSO support, all packets would flow through the non_order register; with SSO, all SSO packet flow through the retry_pkt register, and non_SSO packets continue through the non-order register.

The resend logic basically examines ack status to determine whether a packet is completed (ack_done) or requires retry because of a busy receiver (ack_busy). Ack_done packets return status to the transaction layer through the "ack to command level" block. Non-SSO busy retry packets may be selected to retry through the MUX logic and the 4:1 (or 3:1, depending upon the selected embodiment of the invention) MUX. SSO busy retry packets are enqueued into the retry_pkt FIFO queue. They are selected from this queue through the MUX logic, and the MUX, by the SSO busy retry logic, which is distributed throughout the system and described in the flowcharts for send logic, figures 9-25.

The send logic of Figures 9-25 operates basically in the following manner. Its details of operations will become clearer below in connection with the example flow in relation to the state/flow diagrams of Figures 39 et seq. Each of the logic units (e.g. ack_logic) in the following description refers to those elements in Figure 39.

Figure 9: ack logic for testing packets against
acknowledgments for error conditions, and starting busy retry
in the absence of errors.

Figure 10: ack logic for setting ack logic output for
normal packet processing, in the absence of errors.

Figure 11: handling in the ack logic for non-SSO packets to be loaded into the non_order register.

Figure 12: ack logic handling of the CRC_init_err error retry loop, which resends the last "known good" SSO packet until it receives a valid (done or busy) acknowledgment for this packet.

Figure 13: ack logic handling the CRC_err error retry loop, which resends all packets following the "known good" packet back through the ringlet.

Figure 14: ack logic handling the CRC_ack_chk error retry loop, which tests whether all of the packets retried in the CRC_err loop completed without error or had an error. If any error is found, the loop is retried, beginning with CRC_init_err (fig 30). If no error is detected, CRC_err_end is set.

Figure 15: ack logic handling the CRC_ack_chk error retry loop, which tests the acknowledgment of each packet sent during the CRC_err loop (fig 31) for valid acknowledgment (ack_busy or ack_done) and sets an error bit if any error is detected.

Figure 16: resend logic handling the formatting of packet state information and setting the MUX while executing the CRC_init_err, CRC_err, and CRC_ack_chk loops.

Figure 17: resend logic to detect conditions to complete the CRC_err_end control and set the busy_loop_cmplt control for detecting the completion of the busy retry loop, by testing for retry_pkt queue empty .

Figure 18: resend logic to detect SSO packets as done (ack_done) or requiring busy retry (ack_busy). Busy retry packets are enqueued into the retry_pkt queue; done packets are signalled to the transaction layer.

Figure 19: resend logic to detect non-SSO packets as done (ack_done) or requiring busy retry (ack_busy). Busy retry packets are sent through the MUX for retry; done packets are signalled to the transaction layer.

Figure 20: retry send logic to set the busy loop when no SSO packets are outstanding in the ringlet, and to control the busy retry loop by setting the MUX to select packets from the retry_pkt queue through the retry send block.

Figure 21: MUX logic to detect conditions for starting the busy retry loop, selecting a response packet from the response queue, or selecting a request packet from the request queue.

Figure 22: MUX logic to control the MUX to select a request from the request queue.

Figure 23: MUX logic to control the MUX to select a response from the response queue.

Figure 24: CRC generate logic to format controls and gate the transmission of packets to the serializer.

Figure 25: special purpose modulo 64 comparison logic to perform

greater than (labeled .GT.) and less than (labeled .LT.) comparisons.

What follows is a summary of the flow charts for the receive node, figures 28-38. Reference is made also to Figure 26, Receive Node Logic.

To understand the extensions for this disclosure, one must first form an understanding of the underlying logic. In Fig 26, the fundamental blocks of the receive node in the absence of the SSO ordering extensions would include all blocks with the following deletions and changes:

- Delete seqTable, 26-30.
- Delete seq register, 26-40.
- Delete seq update, 26-120.

Other blocks remain fundamental to the operation of a receive node with no SSO support.

The receive node and send node have some common components with common names. These include:

	send node	receive node
deserializer	7.240	26.80
8/10 decode/CRC check	7.230	26.70
rec_pkt register	7.220	26.60
ack decode logic	7.140	26.140
ack queue	7.130	26.150

Non-SSO Packet Flow

As alluded to above, the present system may be used in either non-SSO or fully SSO environments.

A description of basic (non-SSO) packet flow, and an overview of the figures added for SSO support, follows.

Basic packet flow, non-SSO packets

The role of the SSO ordering logic can be understood from comparison with the basic packet flow for non-SSO packets. Starting with Figure 9, the blocks 9.10, 9.20, and 9.30 inhibit the normal packet handling if the CRC error handling control sequence of CRC_init_err, CRC_err, and CRC_ack_chk are active. If all packets are non-SSO, then the block 9.50, which identifies the packet as non-SSO, sends packet control to figure 11.10.

Figure 11 describes the basic non-SSO error detection logic, comparing the head of the send_pkt queue (see Figures 8 and 39) with the head of the ack queue to see if the acknowledge matches the packet. (Comparisons are done in blocks 11.10 through 11.40.) If the packet has an error, the block 11.50 sets the state "err_ret" to signal this as an error packet. Otherwise, the block 11.60 forwards the "busy" and "done" states from the packet's acknowledge (at the head of the ack queue, 9.130). In both cases, the packet is written into the non_order_reg.

In fig 9.10, describing resend logic for non-ordered packets, the non_order_reg is tested for validity. If valid, the packet is tested for "done" status in block 9.20, and "done" status is sent to the transaction layer in 9.50. If the packet is either "busy" or has "err_ret" state set, then the 4:1_mux is set for resend$ and the packet is forwarded to the send_pkt register, in block 9.60. Finally, in this block the sequence count for non-SSO packets is incremented and assigned to this retried packet.

In Figure 24, the send_pkt register is tested for validity in 24.10 and then forwarded to CRC generation in block 24.30. The pLabel field is defined by the fact that this is a non-SSO packet and by the assigned "seq" field in the same block 24.30. Also, the packet is again enqueued into the send_pkt queue to await an acknowledge packet.

While the flow charts begin with the arrival of packets, the packet is assumed to have been selected by some priority process. In a non-SSO system, this priority may be similar to that of the system of the invention, but all references to "busy" and "retry_pkt queue" would be removed. The selection of either a request or response packet from respective queues in a non-SSO system would be similar to that described in fig 22 and 23.

Logic addition for SSO support

The remaining ack logic, described in the flow charts 9-15, describes extensions for han-

dling SSO ordered blocks.

(Note that the blocks 12.20, 13.20, 14.20, and 15.20 also identify non-SSO send commands in the send_pkt queue (see Figure 39) and branch to the non-SSO packet handling block 11.10. However, these blocks cover the handling of non-SSO packets within a system which has the capability of supporting a mixed programming environment of both SSO and non-SSO commands. That is, these blocks are not a part of a basic non-SSO only environment.)

Similarly, the handling of non-SSO packets in the resend logic block also is described in a single page, Figure 9. The resend logic described in Figures 16-18 is added for SSO support. The retry handling logic, figure 20, is also unique to SSO handling logic.

The MUX logic described in fig 21 to start the busy loop (set in 21.90) is unique to implementations with SSO support. Other logic, selecting request or response packets, would be similar for both SSO and non-SSO implementations. The MUX logic to initiate a request or response packet described in fig 22 and 23 would be similar for both SSO and non-SSO implementations.

Finally, the CRC generate logic in Figure 24 tests the "no_xmit" state bit in 24.40 to determine whether a packet is to be transferred to the serializing logic. Such a gating is unique for SSO support. The logic supporting modulo 64 comparisons is also uniquely required for SSO support.

Fig 26 block diagram overview

Fig 26, like Figures 8 and 39, is a potential block diagram of a receive node (send node for fig 7), used to illustrate how the concepts of the SSO mechanism may be implemented. It is recognized that this is but one of many possible instantiations of a receive (send) node embodying the concepts of this mechanism.

Packet flow in Fig 26 is fundamentally as follows:

- Packets enter the receive node through the deserializer, 26.80, from the ringlet, 26.90. The CRC check and 8/10 decode logic, 26.70, strips any non-packet symbols and loads packets into the 32 bit rec_pkt register, 26.60.

- The start of packet is detected and initial packet decode is

done in this register. Packets utilized by the receive node include:

- send packets (request and response) addressed to this node;
- send packets (req and response) addressed to other nodes;
- acknowledge packets addressed to other nodes.

Send packets addressed to this node are marked for enqueing into the request_in queue (request packets) or response_in queue (response packets), respectively. (Note: the use of queues

Acknowledges to this node are decoded and written to the ack queue, 26.150.

- Receive node packets are written into the test_pkt node. When the start of the packet is in the test_pkt register, the 32 bits with the producerId are in the rec_pkt register. This field indexes the seqTable array for receive node state, including sequence the sequence and accept fields.

—

- The next cycle, the seqTable[producerId] contents are read into the seq register, 26.40, and the pLabel and producerId fields will be in the test_pkt register, 26.50. The corresponding fields are compared in the following blocks for all packets (send packets and acknowledges) in the receive node, although the timing of packet handling depends on the type of packet:

- acpt logic, 26.130, basically determines whether request and response packets addressed to this node are accepted. Both the seqTable and CRC error checking are considered.

- seq update logic, 26.120, determines whether and how the seqTable array state is updated at the end of the packet.

- ack gen logic, 26.110, determines what kind (inhibited on CRC error detection) of acknowledgment is generated, ack done, busy, or error.

- The remainder of the packet is processed (in the case of send packets), with any send packet data enqueued into the request_in (26.10) or response_in (26.20) queues, provided that queue space is available. At the conclusion of the packet, the CRC check completes the operation for the send node.

Figure 27A and 27B show receive packet and acknowledge timing diagrams, respectively. The receive node logic is shown in Figures 28 et seq., and is described as follows.

Figure 28: Detect the start of a send packet (or "command") or an acknowledge packet from header fields in the rec_pkt_reg 26.60 (see Figure 26). For send packets, either requests or responses, addressed to this node, test whether queue space exists in the corresponding input queues, request_in queue 26.10 and response_in queue 26.20, and set control bits accordingly.

Figure 29: Read the seq Table array 26.30 into the seq_reg 26.40. Test for conditions to inhibit validating a packet in the request_in 26.10 or response_in 26.20 queues.

Figure 30: If control bits indicate queue space is available, tentatively enqueued the remainder of a packet addressed to this node in to the corresponding queue, request_in 26.10 or response_in 26.20. Conditions for incrementing control fields or resetting control bits are given.

Figure 31: At the end of a send packet addressed to this node, depending on the CRC check, gen-

erate an acknowledge packet to the send node and validate the packet in the request_in 26.10 or response_in 26.20 queue.

Figure 32: Generate tentative acknowledge conditions (done, busy or error) for use in acknowledge generation on Figure 31, depending on CRC error checking.

Figure 33: Continuation of tentative acknowledge generation (done, busy, or error) for SSO packets with valid sequence.

Figure 34: Test for conditions to tentatively set update conditions for the seq Table array at the conclusion of the packet, or to inhibit update. Update will depend on valid CRC check.

Figure 35: Continuation of tentative sequence update generation in the case where the packet is SSO ordered with valid sequence.

Figure 36: At the conclusion of the send packet, test for valid CRC in 26.70. If the CRC is valid, and the seq Table write is not inhibited, update the seq Table.

Figure 37: Generate seqTable update fields for acknowledge packets addressed to other nodes.

Figure 38: Continuation of seq Table update field generation for acknowledge packets addressed to other nodes, for the case where packets are SSO ordered with valid sequence.

The example of Figures 39 et seq. demonstrate and exemplary operation of the system of the invention, with the state and actions described in those Figures, which conform to the logic in the foregoing figures. Figures 39 et seq. show, step by step, the state of the producer node for each transfer of packets among the registers and queues, and by inspection of those diagrams, it will be seen that the system does accommodate SSO, nonidempotent commands in the case of errors in connection with a busy retry loop and an ill-behaved node.

In reading the logic/flow diagrams in the figures, various states are referred to. These are

defined below. In addition, Appendix A attached hereto includes a glossary of the terminology and variables used in connection with the current design.

State	Interpretation
i	in_proc
b	busy (variations: ack_busy, busy loop retry)
r	retry in CRC loop
b1	first packet in busy retry loop
r1	first packet in CRC retry loop
x	either ack or busy
d	ack_done
BRC	busy resend loop
CRC	CRC err resend loop
SRL	send_pkt recirculating loop

Discussion of the Preferred Embodiment of the Invention in Light of Proposed 1394.2 Standard

The following presents an overview and summary of a proposed mechanism to support fully pipelined packet initiation in a ringlet topology based on IEEE P1394.2, where the pipelined packets may include send packets and response packets from address spaces requiring strong sequential ordering (SSO) and support for non-idempotent commands. This mechanism can be extended over arbitrary P1394.2 switch topologies by ensuring that every communication link in the topology, in turn, maintains SSO ordering.

The proposed mechanism supports SSO and non-idempotent commands in a fully pipelined mode of operation for non-busy, non-error packet transmission, while tolerating both CRC errors and busy receiver conditions through an efficient hardware retry mechanism. It supports a variety of resend policies for mixing new packets with retried busy packets. It enables a "correctable error" kind of error handling policy, since CRC error packets may be retried to any programmable limit. This capability, in turn, enables a "preventative maintenance" function for nodes reporting successful retry operations of CRC error packets.

Also note that the use of SSO, non-idempotent capability within a bridge significantly simplifies the design of bridges between IEEE P1394.2 and other protocols that have larger (>64 B) packet formats. For example, a write command for a 2 KB block from an IEEE 1394-1995

node, which expects a single response packet from the receiving node at its conclusion, can be implemented as a series of move commands (no response) and a concluding write, with the guarantee that the operation of the IEEE 1394-1995 packet will be preserved over the bridge.

The cost of SSO ordering is 2 bytes of state per node supported on the ringlet, plus state machines, a separate register for non-SSO packets, muxes, and a 6 bit comparator. The flow description is now based on simple FIFO queues and registers, rather than more complicated array structures. Since the maximum configuration for a ringlet is 63 nodes, this bounds the required state to support SSO to ~128 B max (plus pointers and state machines). "Profiles" of smaller supported configurations can further reduce the cost by cutting the supported ringlet node count.

Brief overview of IEEE P1394.2

IEEE P1394.2, or Serial Express, is a proposed extension of IEEE 1394-1995, or Serial Bus, to Gigabit+ transmission levels. Basically, the protocol supports the fundamental command set of IEEE 1394-1995 while rearchitecting the transfer protocol to emphasize low latency and high throughput with 16 B and 64 B packets.

IEEE P1394.2 is based on an insertion buffer architecture with a ringlet topology. What this means is that a each node includes two unidirectional wires, one for packets and idles (for synchronization and flow control) coming in, and one for packets and idles going out. A bypass buffer shunts packets and idles coming in over to the outgoing wire, possibly with delays. A single new send or response packet (defined below) may be inserted by the node into the ringlet if its bypass buffer is not full; this is the "insertion buffer" architecture. Any packet that enters the bypass buffer while the node is inserting the new packet is delayed until the new packet is sent. The small packet sizes (<= 64B) make this architecture both possible and efficient.

The packet format includes a 16 bit field for packet routing usage plus a 48 bit extended address. Each node supports full duplex operation over a single cable with separate send and receive paths. When several nodes (up to 63) are interconnected, initialization software breaks redundant loops and configures those nodes as a single ringlet. Multiple ringlets (with as few as 2 connections) can be interconnected through switches, up to a maximum topology of 16K nodes.

The IEEE P1394.2 protocol supports four modes of operation, based on two address modes (directed and multicast) and two service types (asynchronous and isochronous). The SSO ordering is of particular interest for asynchronous, directed service, which includes (read, write,

move, and lock) operations, but the proposed mechanism is not limited to only this mode of service.

SSO Ordering: Issues

In the absence of SSO ordering support, packets from any node to any destination have no constraints with respect to one another. With both CRC errors and busy receivers to contend with, this means that any two packets may arrive at the destination node in a different order from that sent, if at all.

This uncertainty is typical of networked environments and is dealt with using a variety of software-based protocols. One reliable mechanism, for example, that can be used with IEEE 1394-1995 to serialize an interrupt generating write request with a prior data packet, is to send a write command (which requires a response from the destination) for a large, say 2KB packet, and then send the write command to generate the interrupt after the response for the data has come back.

Such mechanisms, when extended into a domain with much smaller (<= 64B) packets, a domain that emphasizes low latency transfers, can become both complicated and inefficient. The software overhead for establishing ordering and data transfer completion both detracts from available compute cycles and adds directly to user process - to - user process latency.

The SSO extension with P1394.2

The proposed SSO extension depends on these fundamental assumptions for the operation of a ringlet:

Each command in (asynchronous, directed) mode is executed by the originating (sender) node generating a send packet and, depending on the command, the ultimate destination node (possibly) generating a response packet. The "send" packet is removed by a single receiving node on the sender's local ringlet and replaced with an "acknowledge" packet from that local receiving node back to the sending node. The local receiving node may be either the final destination for the send packet or an agent that forwards the packet, perhaps through a switch, to its destination. Depending on the type of command, the ultimate destination node may send a "response" packet, which is also captured by some node on its local ringlet and replaced by an "acknowledge" packet

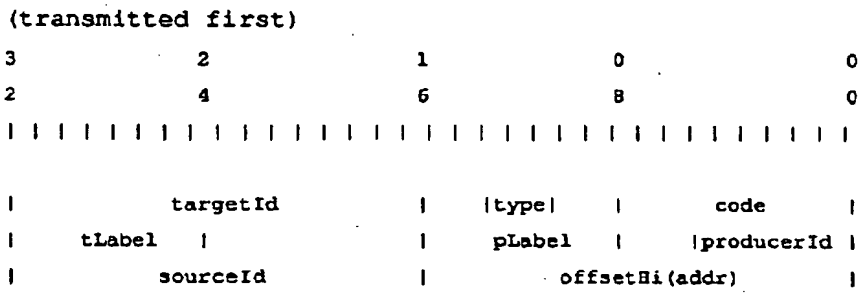
back to the "response" generator.

In the discussion below, the term "producer node" is sometimes used to indicate the node that initiates the packet, whether that node is a sender node initiating a send packet, a receiver node initiating a response packet, or a bridge or switch node forwarding a send or response packet.

The target for both send and receive packets is identified by a global, 16 bit "targetId" field. In the absence of errors, some unique node on the ringlet, perhaps a bridge or switch node, will recognize the targetId address and strip the packet. A "sourceId" field uniquely identifies the global address of the sending node.

Other fields, which are included within the packet for returning local ringlet acknowledgment, are fundamental to the proposed SSO ordering enhancement. These include the (6 bit) "producerId" field, which identifies the nodeId that originated this packet **on its local ringlet**, and the (8 bit) "pLabel" field, which is assigned by the "producerId" node to uniquely identify this packet. Both the producerId and pLabel fields have meaning only within a local ringlet and will be reassigned, for example, if the packet is forwarded through a switch. Similarly, the "producerId" and "pLabel" fields are also reassigned by the destination node when it sends a response packet.

Other fields mentioned below include the "type" field (which distinguishes types of commands and acknowledgments); the "code" field (command identification); the global "tLabel" field, which is returned in a response packet to uniquely identify that response to the sourceId node; and the crc32 field, a 32 bit CRC code that covers both the header and data for the packet.



```
|                          offsetLo(addr)|

(                          optional data                          )

|                              crc32                              |
```

The acknowledgment packet generated by the receiver node echoes some of the fields from the send or response packet back to the "producer node", so that it can uniquely recognize the packet being acknowledged:

```
(transmitted first)
3              2              1              0              0
2              4              6              8              0
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|     localBus       |producerId |   |type |    |     pLabel       |

      |                       crc32              |
```

To create the acknowledge address, the "targetId" address field of the original packet is replaced by a "localBus" identifier (all 1's, typically), plus the 6 bit producerId field from the original packet. The pLabel field uniquely identifies the original packet to the "producer node". The "type" field encodes the type of acknowledgment. The crc32 field for the acknowledge packet may be replaced by some other error detection code without loss of generality for the SSO mechanism described here.

Fundamental to the proposal is the assumption that local ringlet transmission is unidirectional and bypasses no nodes. That is, when a node sends either a send packet or a response packet, then every node on the subring will see either that packet or the acknowledgment for that packet flow through the node's bypass buffer.

This assumption underlies the basic operation of the P1394.2 ringlet (although in a proposed P1394.2 option, a "short cut" routing feature would not support SSO ordering). As a result of this, for every send response packet that is transmitted, each node on the ringlet can observe both the producerId and the pLabel fields for either the send/response packet or its acknowledg-

ment.

The P1394.2 protocol supports split response transactions, where a response is required. Packets for (asynchronous, directed) service are subject to two conditions that may inhibit packet delivery:

the packet - or its acknowledgment - may suffer a CRC
error or other detectable error;
the packet may be rejected by a "busy" receiver at
the destination node.

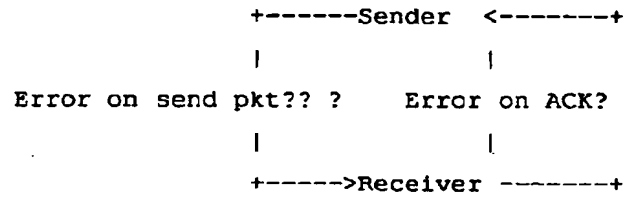
In the case of a CRC error, the only reliable detection is at the originating node, which will detect the error if the expected acknowledgment has not been seen within two ringlet revolutions. (A bit is circulated by a "scrubber" node that is complemented at every ringlet revolution to assist in local node timeout detection.) The exact method of timeout detection employed is not fundamental to the operation of the SSO ordering support mechanism described here.

In the case of a "busy" receiver, the receiver node substitutes an ACK_busy acknowledgment packet for the send or response packet in (asynchronous, directed) service.

System overview: known good information at a node

The claim that the pLabel and producerId fields can be used to implement an SSO ordering mechanism in a ringlet topology depends on certain information being unambiguously known to both the producer node and the receiver node. Here is a summary of this information.

The basic concept here is that each packet may have two uses in the ringlet: to transmit commands or data in the SSO program space, and to transfer ringlet sequence state information. Only the sender knows the sequence number of transmitted packets, and only the receiver knows the last valid, accepted packet. Pictorially:



Producer nodes:

- Send/response packet fields to control SSO operation in ringlet nodes: The pLabel field for send and response packets from each producerId is used to convey three subfields (details later):

```
pLabel.sso (1 bit)
pLabel.bzseq (1 bit)   /* Unused for non-SSO space */
pLabel.seq   (6 bits)
```

The "pLabel.sso" bit defines, dynamically, whether the address space for this packet is SSO ordered (sso = 1) or not (sso = 0). The "bzseq", or busy sequence bit has meaning only in SSO ordering space and is used to help a receiver node determine whether it must issue a busy retry acknowledgment to a send / response packet. The 6 bit "seq" field is a wrapping, modulo 64 "counter" used by all local ringlet nodes for identifying valid packet sequences from this producerId.

The maintenance of SSO ordering between each producerId node and the other nodes on the ringlet depends on the producerId "metering" its send and receive packets and setting their pLabel fields to ensure that each other node can maintain proper state information. By comparing that state information with incoming pLabel fields from the producerId, packets can be properly dispositioned to maintain SSO ordering.

("Metering" requires that any producer node may have no more than 32 packets "outstanding" in its local ringlet, based on the 6 bit "seq" field. "Outstanding" means counting from the current packet to send back to either the earliest prior packet from this node that generated a "busy" acknowledgment, or the earliest prior packet for which no acknowledgment has been received, whichever is greater. There is no architected limit to the number of packets that may be outstanding beyond the local ringlet, for example awaiting responses at destination nodes.)

Under normal operation in SSO space, the pLabel.seq field is incremented by 1, modulo 64, with each new send/response packet or with each retried busy packet. If a CRC error is

detected, the SSO state for all local ringlet nodes must first be resynched by resending some packets (below).

The term "CRC error" used below may be generally understood to include not only CRC errors, but any non-corrected error condition for a packet (send, response, or acknowledge) that can be detected by the sending node.

CRC error loop reset: Once a CRC error is detected at the producer node, the SSO state values maintained for this producerId at every node in the ringlet are in question. Each ringlet node's SSO state information for this producerID must be reset, without any change in either the SSO programming space for this producerId or for any other producer node.

Two important features of the mechanism are used to accomplish this:

1. Each receiver node on the ringlet maintains an array of SSO state, indexed by the local producerId for each sending node on the local ringlet. The SSO state contains control bits plus two sequencing fields. One of these fields is used to determine whether a current ringlet packet addressed to this receiver has a valid SSO sequence number, and the other field is used to determine, along with the state of the node's receive buffer, whether the packet may be accepted or rejected by the node and what type of acknowledgment will be returned (done, busy, or error).
2. Each producerId that detects a CRC error has the ability to retry a sequence of SSO ordered packets, beginning with the last good SSO ordered packet prior to the error packet, and extending through the last packet sent from this node prior to the detection of the error. The known good SSO ordered packet is retried (if necessary) to begin the CRC error loop, in order to establish a valid sequence base count in the ringlet. Then the remaining packets are resent, up through the last packet before the CRC error. If any error is detected in this retry sequence, the loop is repeated until either it succeeds without error, or the error retry loop concludes without success; in such cases, higher level protocols may deal with the error. Once this sequence completes without error, all nodes in the ringlet will have their SSO state restored to consistent, known good values.

Receiver nodes:

A receiver node can detect a valid packet sequence from any producerId if its SSO state information is correct. This state is initialized by the producerId on reset and then re-initialized if a CRC error is detected by the producerID.

A receiving node may or may not detect a CRC error send packet or acknowledge. Even if it does, since the packet's contents are unreliable, no node state information can be updated. It is the sending node's responsibility to detect a timeout and resend any CRC error packet.

However, the receiving node can detect an out-of-sequence transmission of a valid ordered packet following a CRC error packet from some producerID. Each such out-of-sequence packet must be detected and rejected at the receiver while being acknowledged as ACK_error. The node sending this packet marks the packet and retries it in its error loop.

A packet may be rejected in asynchronous operation if the receiving node is busy or, with SSO ordering support, if the packet is a retransmission of a packet that had been previously accepted. In SSO ordered mode, once a first packet from a given producerId has been rejected as busy, then all subsequent ordered-packets from this producerID must also be rejected as busy until the first busy packet is retried.

The state of the receiver for SSO must then support the acceptance of new packets into a non-busy receiver buffer and the rejection of previously accepted packets that are resent for error recovery and any new packet sent following a "busy" rejection. Since the acknowledge packet returned to the producer is subject to error, the receiver state must be able to reproduce the ack_done and ack_busy packets if any packet is retried.

To accomplish this, each receiver node requires a state array, called seqTable, indexed by the producerId, with the following six fields:

```
seqTable.verif [producerId] (1 bit)
seqTable.sso [producerId] (1 bit)
seqTable.seq [producerId] (6 bits)
seqTable.bzseq [producerId] (1 bit)
seqTable.busy [producerId] (1 bit)
seqTable.acpt [producerId] (6 bits)
```

In SSO operation, the seqTable fields are used as follows. The "verif" field indicates both that this seqTable has been initialized and that the producerId is a verified source. The "sso" field defines the (static) capability of this producerId node to support SSO ordering. (By comparison, the pLabel.sso bit is a dynamic bit set according to the address space ordering for the producer node.) The "seq" field is compared with the pLabel "seq" field for packet sequencing by the receiver node. The "bzseq" bit is compared with the corresponding pLabel "bzseq" bit for SSO packets to identify the retransmission of first busy packets. The "busy" bit is set on the first busy acknowledgment of an SSO packet to the producerId and (possibly) reset when this busy packet is retried, depending on the receiver buffer. Finally, the "acpt" field is used in both the acceptance and rejection of SSO packets and in determining whether to acknowledge as "done" or "busy". These determinations require comparison with the incoming packet's "seq" field.

The receiver node can detect a valid new, SSO ordered packet from a given producerID if:

```
pLabel.seq = seqTable.seq[producerId] + 1
```

If the comparison is valid, the seq field is updated:

```
seqTable.seq[producerId] = pLabel.seq
```

For other comparisons, see below.

The seqTable.seq[producerId] field is reset to the current pLabel.seq value when this node detects that this producerId is initializing a CRC error retry loop. This is indicated by comparing the seq fields for a packet from - or acknowledge to - this producerId:

```
if (pLabel.seq <= seqTable.seq[producerId]) then
    seqTable.seq[producerId] = pLabel.seq
```

Only the seqTable.seq value is modified, not the seqTable.acpt[] field, during the initialization of the CRC error loop. The packet will be rejected, but using the same accept/reject logic nor-

mally used (below). It is important that the seqTable state information be used to correctly acknowledge all retried packets as "done" (meaning previously accepted) or "busy"; the logic is described below.

If the "seq" comparison is valid, then packets are accepted or rejected according to this comparison:

```
accept: {  {pLabel.bzseq ~= seqTable.bzseq[producerId]}
           & {receiver_not_busy}
        | {pLabel.bzseq = seqTable.bzseq[producerId]}
           & {seqTable.busy[producerId] = 0}
           & {receiver_not_busy}
           & {pLabel.seq > seqTable.acpt[producerId]} }

reject: {  {receiver_busy}
        | {pLabel.bzseq = seqTable.bzseq[producerId]}
           & {seqTable.busy[producerId] = 1}
        | {pLabel.bzseq = seqTable.bzseq[producerId]}
           & {pLabel.seq <= seqTable.acpt[producerId]} }
= ~ accept
```

Less formally, packets are eligible for acceptance if a busy retry loop is being started and the receiver buffer is not busy, or if a busy retry loop is not being started and the "busy" state is reset and the receiver buffer is not busy and the sequence comparison shows that the producer node is not in an error retry loop. Packets are rejected if not accepted.

If the "seq" comparison is valid, then packets are acknowledged "done" or "busy" according to this comparison:

```
ack_don: {  {pLabel.bzseq ~= seqTable.bzseq[producerId]}
          & {receiver_not_busy}
```

```
        | {seqTable.busy[producerId] = 0}
          & {receiver_not_busy}
        | {pLabel.seq < seqTable.acpt[producerId]} }

ack_bzy: {  {pLabel.bzseq = seqTable.bzseq[producerId]}
          & {seqTable.busy[producerId] = 1}
          & {pLabel.seq >= seqTable.acpt[producerId]}
        | {pLabel.seq >= seqTable.acpt[producerId]}
          & {receiver_busy}
        = ~ ack_don
```

Less formally, packets are sent ack_don if a busy retry loop is being started and the receiver is not busy, or if the "busy" state is reset and the receiver is not busy, or if the sequence comparison shows that in an error retry loop, the packet being retried had been sent before the first "busy" packet. Packets are acknowledged as ack_bzy if they are not acknowledged as ack_don (again with the assumption of valid sequence comparison).

- The first time that a receiver node rejects a packet as busy from a given producerId, it must change its "busy" state by setting (seqTable.busy[producerId] = 1). This has the affect of freezing the seqTable.acpt[] field at the "busy" packet's sequence number. This field remains frozen until the first busy packet is retried.

Conditions for setting "busy" and "acpt" fields depend on whether a busy retry loop is being executed by the producer node. If the "seq" comparison is valid, then the first packet in a busy retry loop can be detected from

```
pLabel.bzseq ~= seqTable.bzseq[producerId]
```

(Note that the "bzseq" field is not changed while executing an error retry loop. Since this is true, it will always be true that

```
{pLabel.seq > seqTable.acpt[producerId]}
```

in the packets of a busy retry loop.)

If a first retry packet is detected, then seqTable entries are set a follows:

```
{ seqTable.bzseq[producerId] = pLabel.bzseq
  seqTable.busy[producerId] = receiver_busy
  seqTable.acpt[producerId] = pLabel.seq  }
```

Note that the first retry packet will be rejected by the comparison criteria above and acknowledged as either "done" or "busy". If a packet is not a first retry packet, this is detected from

```
pLabel.bzseq = seqTable.bzseq[producerId]
```

Then seqTable entries depend on packet acceptance:

```
if {  (seqTable.busy[producerId] = 0)
    & (pLabel.seq > seqTable.acpt[producerId]) } then
     {seqTable.busy[producerId] = receiver_busy
      seqTable.acpt[producerId] = pLabel.seq
     {
else /* previous busy packet or error retry loop*/
     {no update
     }
```

Finally, each node can detect a packet with a sequence error from the producerId if:

```
pLabel.seq > seqTable.seq[producerId] + 1
```

This can happen if this packet has valid CRC but some prior packet did not. (The count comparisons are not full 6 bit arithmetic comparisons; see details below.) No seqTable entries are

updated for packets with sequence errors.

Ringlet SSO state initialization:

- The producer node must ensure that the state information at each node on its ringlet is initialized before sending actual SSO send/response packets.

A "trusted producer" policy may be supported with the proposed seqTable state bits and write send packets to initialize the seqTable. Three steps are needed:

1. The seqTable[] array is powered on or reset with seqTable.verif[] = 0 for all producerId's.
2. Through an (unspecified) process in a non-SSO address space, the local node verifies a proposed producerId and then writes its own seqTable.verif[producerId] = 1. The "verif" bit may only be written by the local node.
3. With seqTable.verif[producerId] activated, the producerId sends a write (or move) command addressed to this node's seqTable. The packet is identified as an initialization packet through its target address (seqTable).

The values written into the seqTable are taken from the initialization packet's data as follows:

```
if (seqTable.verif[producerId] == 1) then
     {seqTable.sso[producerId] = 1
      seqTable.seq[producerId] = [current "seq"
from this producerId]
      seqTable.bzseq[producerId] = [current "bzseq"
                          from this producerId]
      seqTable.busy[producerId] = 0

      seqTable.acpt[producerId] = [current "seq"
                          from this producerId]
```

}

Each node in the ringlet must be verified and initialized separately.

A policy like the above can be utilized to dynamically add (or subtract) nodes from an existing ring as well as initialize a newly powered-on ring.

A policy like the above can enable selective "trusted node" exchanges. For example, node A can exchange with node B, node C with node B, but A and B can be prevented from SSO exchanges.

Application space

The proposed mechanism can operate in a generalized 1394.2 network topology which includes:

nodes that are SSO-ordering capable and those that are not (static capability); and

nodes that support multiple address spaces that may be either SSO ordered or not SSO ordered (dynamic capability).

Note that dynamic capability is important since within IEEE P1394.2 there are modes (isochronous) that may not be SSO ordered, and bridging to non-SSO ordered spaces (such as IEEE 1394-1995) must be supported. Generally, it is presumed (not required) that the software interface to the IEEE P1394.2 node is a set of address spaces with potentially varying programming models. At least one of these models is presumed to be an SSO model supporting non-idempotent commands.

The application of the SSO ordering mechanism is described here as it applies to the IEEE P1394.2 (asynchronous, directed) service. This does not mean that it cannot apply to other modes as well. This mechanism supports an arbitrary stream of commands from multiple address spaces, meaning that SSO commands may be interspersed with commands from other programming models.

If SSO ordering is to be maintained, end-to-end, from an SSO address space (or SSO domain) to a destination node, then it is clear that the destination node for SSO domain com-

mands must be SSO-ordering capable, and that all intermediate nodes in switches or bridges must be SSO-ordering capable.

IEEE P1394.2 Changes required for support

The proposed SSO support mechanism can be implemented with the current 0.6 version of the spec without modification as an undocumented optional feature. When patents covering this mechanism are filed, however, it is planned that this mechanism will be added to the IEEE P1394.2 documentation as (at least) an optional feature.

While there is no required change, one encoding addition is suggested for clarity and another encoding addition is needed for support of cost-effective bridging to IEEE 1394-1995 nodes.

The encoding addition for clarity is the addition of an "ACK_error" acknowledgment to the "type" field, replacing a reserved code:

```
Type field:0  Normal, non-SSO
              1    Extended, non-SSO
                 2[reserved]
              3    [reserved]
              4    ACK_done
              5    ACK_busy
              6    ACK_more /* Multicast mode retry req'd */
                *7ACK_error*/* Sequence error in SSO mode */
```

The encoding addition for bridging is specifying bits to delineate the starting, middle, and ending 64 B packets whose destination is reached through an interconnect that supports longer packet types. For example, 2KB packets over Ethernet or 1394 can be broken up into three types of packets:

Start packet: possibly a partial block, aligned at its end on a 64B boundary.

Middle packet: always 64B, always aligned.

End packet: possibly a partial block, aligned at its start on a 64B boundary.

A large packet entering a P1394.2 ringlet through a bridge may be broken up into smaller packets (for example, a large write packet broken into multiple 64B moves plus a 64B write, for example). But reassembling a large packet from even SSO ordered 64B packets, for example to bridge to a 1394 destination node, requires that the component parts of the large packet be easily identified. The need is particularly strong because of the typically low inefficiency of protocols designed for large packets when dealing with small 64B packets. A start/middle/end encoding, when combined with the SSO ordering capability, allow a bridge to very efficiently reassemble packets for delivery.

Retry overview

Two types of retry conditions must be dealt with for SSO ordered address spaces supporting non-idempotent commands: "busy" conditions at the receiving node and CRC error conditions detected in the local ringlet. In the latter case, it is assumed that "CRC error" means the corruption of the packet (either the send packet, the response packet, or the ACK to either) so that neither source nor destination fields represent reliable data. The term "CRC error" should not be construed to mean restriction to only CRC error codes, but more generally should include any error detecting mechanism. For efficient operation of ordered domains, is assumed that "busy" retry conditions are relatively infrequent but still much more frequent than CRC error conditions. However, in no case is the validity of the proposed scheme dependent on the frequency of occurrence of either "busy" or CRC error cases.

For "busy" retries in ordered domains, in the absence of errors, it is only necessary to retry packets to those specific domains which respond "busy". The proposed scheme treats "busy" conditions in alternate destination nodes as independent; if node A transfers a sequence of packets to both nodes B and C, and only B responds "busy", then only the packets to node B are retried.

CRC error packets (either send/response packets or ACKs) are ultimately detected at the sending node. In SSO ordered domains, this detection might be either as a "sequence error" (an ACK_err response is detected before a time out) or as a ringlet time out (two revolutions with no acknowledgment). In SSO ordered domains, receiver nodes have the responsibility of sending an ACK_error to reject each send/response packet following a CRC error occurrence if a sequence error is detected from some node, but otherwise must maintain no error state information. When the sending node does detect that a CRC error has occurred in either its packet or an ACK, then it

must retry all outstanding SSO ordered domain packets from the point of the last valid, completed packet up through the last outstanding packet sent before the error is detected and packet transmission is halted. The policy for retrying CRC errors in non-SSO domains is unspecified here.

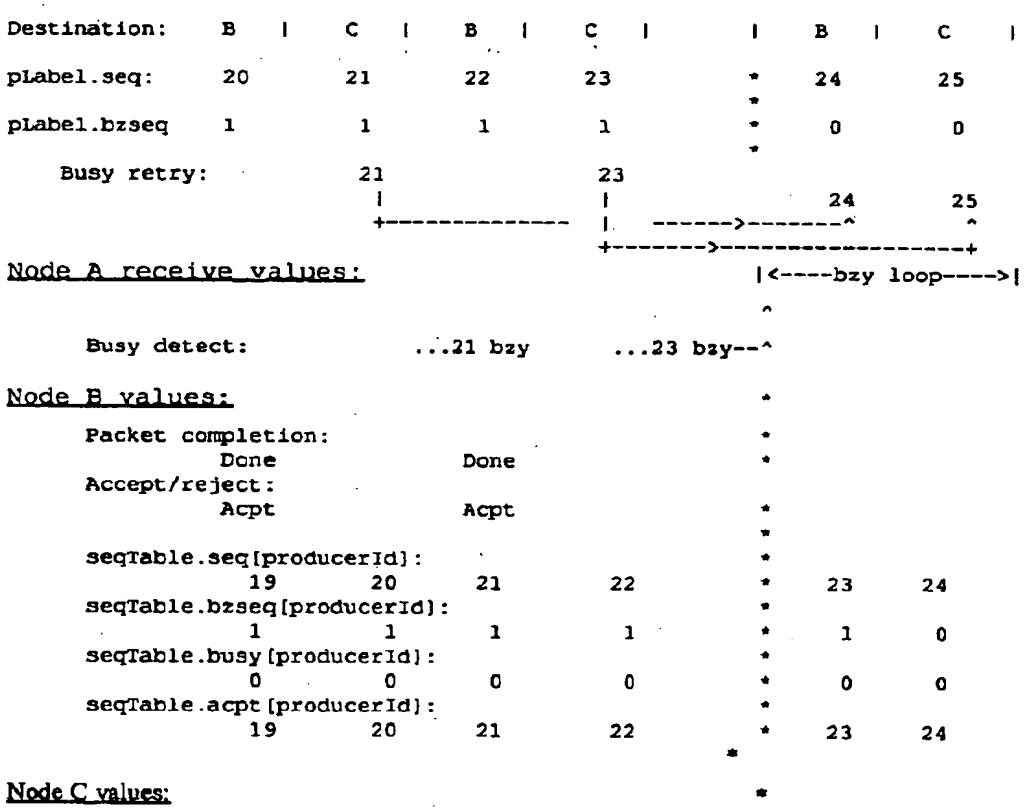
Busy retry example

The proposed mechanism for SSO ordering support is illustrated with an example before detailed descriptions are given, in order to better visualize how the mechanism would work.

]

Case 1: Busy retry from node A to node C:

Packet sequencing from standpoint of the send packet:



Packet completion:						
Accept/reject:	Bzy Rej		Bzy Rej		Done Acpt	Done Acpt
seqTable.seq[producerId]:						
19	20	21	22		23	24
seqTable.bzseq[producerId]:						
1	1	1	1		1	0
seqTable.busy[producerId]:						
0	0	1	1		1	0
seqTable.acpt[producerId]:						
19	20	21	21		21	24

Some observations about this example:

Packets are generated through the send port, but the busy acknowledgment is detected asynchronously in the receive port. The busy retry loop requires that all outstanding prior transactions be acknowledged prior to retransmitting the first retried busy packet (24). Also, the decision on when to retry busy packets may depend on other implementation considerations, such as the arrival rate of new transactions. In this example, packet 23 is the last packet sent before the busy loop resends packets 21 and 23 as renumbered (pLabel.seq) packets 24 and 25.

Only packets marked busy (in this case, only those to node C, beginning with packet 21) are retried; there is no need to retry completed packets.

New sequence numbers are assigned by node A to the retried packets.

Once a first "busy" packet is acknowledged at node C (packet 21), then every packet with the same or higher sequence number addressed to node C (packet 23 in this example) must be rejected with "ack_busy" until the first "busy" packet is retried (indicated by a change in "bzseq"). In an error retry loop, it is possible that a packet prior to 21 is retransmitted. In this case, the packet would be rejected and acknowledged as "ack_done" to prevent its being (incorrectly) retried. Every other node that acknowledges a first "busy" must behave the same way (not shown in this example).

SSO ordering is maintained, pairwise, from the sending node to each of its potential

receiving nodes. However, if node A send SSO ordered packets to node B, and A also sends SSO ordered packets to node C, it does **not** follow that the relative ordering packet arrival between nodes B and C must be maintained for pairwise SSO correctness.

The beginning of a busy retry loop by this producerId node is signalled to all ringlet nodes by complementing the pLabel.bzseq field (from 1 to 0). This swap can be detected at each ringlet node by comparing its seqTable.bzseq[producerId] value with the incoming pLabel.bzseq; if they disagree, then the seqTable value is rewritten:

```
seqTable.bzseq[producerId] = pLabel.bzseq.
```

Also, if the retried busy packet is accepted in the receive buffer, then the "busy" bit is reset:

```
seqTable.busy[producerId] = 0.
```

For all other (non-receiver) nodes, seqTable.busy[producerId] is set = 0.

CRC error retry examples

Two examples of CRC error retry without busy acknowledgments are followed with an example of a CRC error occurring around a busy retry loop.

Case 1: CRC error on send packet to node C, detected at node C.

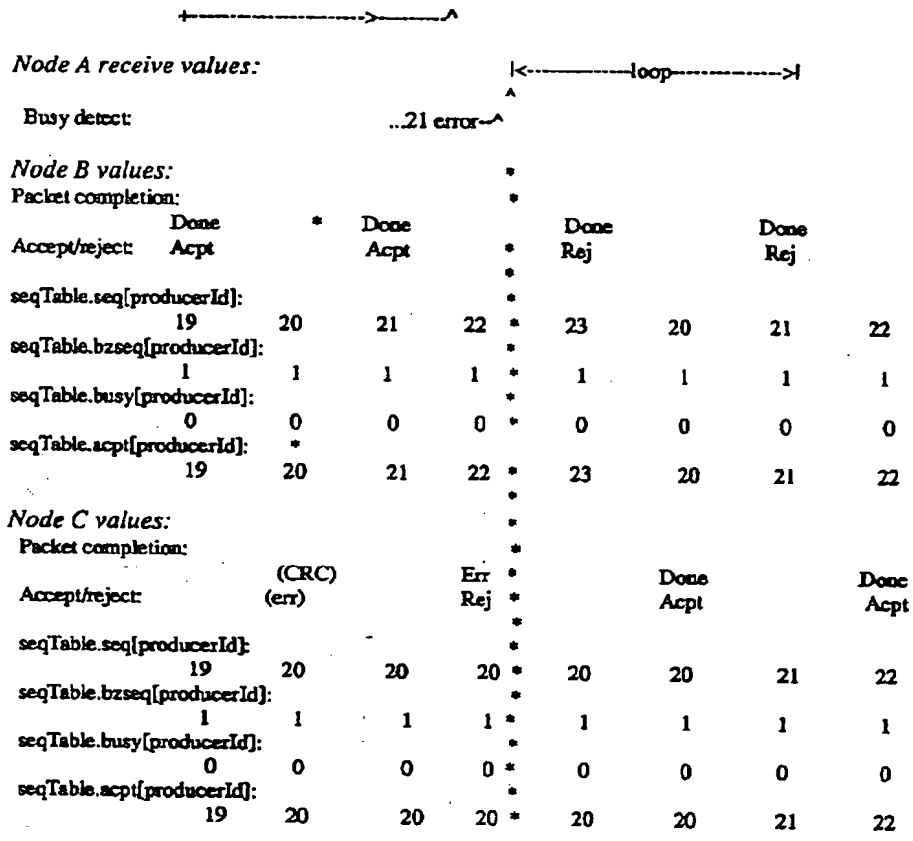
Packet sequencing from standpoint of the send packet:

Case 1: CRC error on send packet to node C, detected at node C.

Packet sequencing from standpoint of the send packet:

Node A send values:

Destination:	B	C	B	C	B	C	B	C
pLabel.seq:	20	21	22	23	20	21	22	23
pLabel.bzseq	1	1	1	1	1	1	1	1
CRC err retry:	20				20			



Some notes on this example:

The detection of a CRC error occurs within the receive node, asynchronously from the send node. In this example, packets 22 and 23 are transmitted before the CRC error for 21 is detected.

Because of the many locations that a CRC error can occur, different nodes may detect - or fail to detect the error in different ways. In this example, the CRC error occurs after node B, so it never sees any sequence error. Node C, on the other hand, fails to detect packet 21 and then detects that subsequent packets/acks 22 and 23 have sequence errors, since its last recorded sequence value is 20.

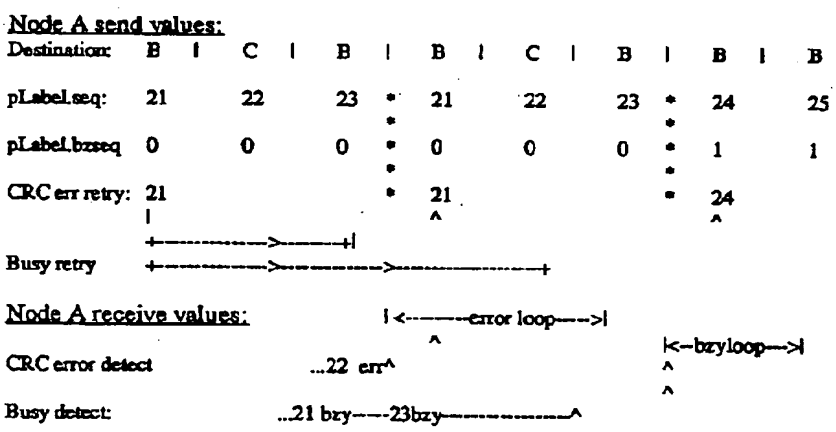
Once a CRC error is detected at the sender, its CRC error loop must first retransmit the last good packet before the error (packet 20), repeatedly if necessary, to reestablish the sequence count (to 20) within the seqTable.seq[producerId] field of each local ringlet node. Based on comparison with the receiver node's seqTable.acpt[producerId] field, this packet will always be rejected, regardless of the target. Along with the sequence number for the known good packet, the producerId also sends out the current bzseq value in pLabel to set to a known good state every seqTable.bzseq[producerId].

While the seqTable.bzseq[producerId] value is reset, each node's seqTable.busy[] and seqTable.acpt[] fields must be preserved. They represent the state necessary to accept or reject packets in the CRC error retry loop with the correct ack done or busy response for this destination. In this example, node B rejects send/response packet 22, while node C accepts as new both packets 21 and 23.

The sender must resend all packets that had been transmitted up to the point that it halted new packets, in this case through packet 23. With no errors, nodes B and C track the sequence numbers.

Case 2: CRC error on acknowledge packet from node C, detected at node A.

Packet sequencing from standpoint of the send packet:



Node B values:

Packet completion:								
	Bzy		Bzy	Bzy		Bzy	Done	Done
Accept/reject:	Rej		Rej	Rej		Rej	Acpt	Acpt
seqTable.seq[producerId]:								
	20	21	22	23	21	22	23	24
seqTable.bzseq[producerId]:								
	0	0	0	0	0	0	0	1
seqTable.busy[producerId]:								
	0	1	1	1	1	1	1	0
seqTable.acpt[producerId]:								
	20	21	21	21	21	21	21	24

Node C values:

Packet completion:								
		(CRC)			Done			
Accept/reject:		---			Acpt			
seqTable.seq[producerId]:								
	20	21	21	21	21	22	23	24
seqTable.bzseq[producerId]:								
	0	0	0	0	0	0	0	1
seqTable.busy[producerId]:								
seqTable.acpt[producerId]:								
	20	21	21	21	21	22	23	24

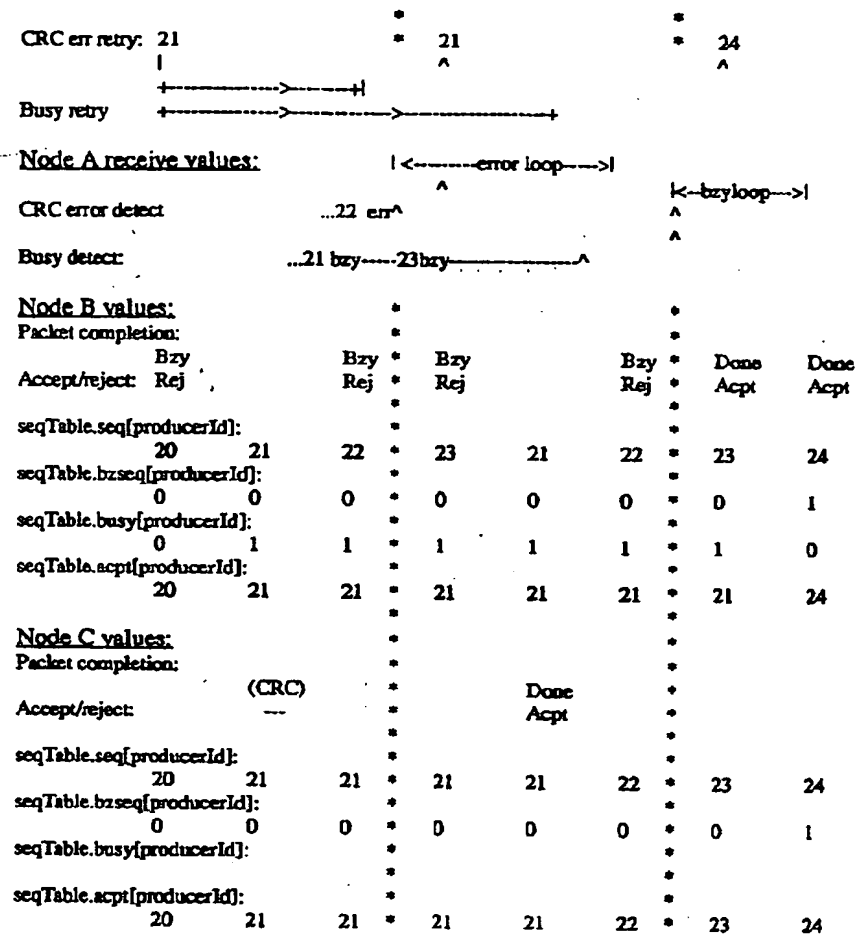
This example illustrates the inability of the sender node to detect whether a packet has been accepted at its destination. Since node C in this example has seen no error, both packets 21 and 23 have valid pLabel.seq fields and are given ACK_done responses.

Case 3: CRC error on send packet to node C, detected at node C before beginning busy retry loop for node B.

Packet sequencing from standpoint of the send packet:

Node A send values:

Destination:	B	C	B	B	C	B	B	B
pLabel.seq:	21	22	23	21	22	23	24	25
pLabel.bzseq	0	0	0	0	0	0	1	1



This example is based on the following sequence of events:

Node B is sent 2 send/response packets, 21 and 23, which it rejects as busy.

Node C is sent a packet 22 which it never sees because of a CRC error at its node; no error is detected at node B.

Node A begins its busy retry for node B packets by first waiting for the completion of outstanding Acks. Now node A detects the CRC error. It begins a CRC error loop, which must include all packets from 21 (last good packet before the error) through 23 (last

packet sent before the error is detected).

In this example, when the CRC error retry loop is entered, retried packets to node B in the error loop are rejected based on the pLabel.seq value and node B's seqTable.busy[] and seqTable.acpt[] fields. Since the seqTable.busy[] bit is set, packets are rejected; since seqTable.busy[] is set but both packets had originally been marked "busy" (based on comparison of the pLabel.seq value with the seqTable.acpt[] field), both retried packets 21 and 23 are acknowledged as "busy".

The seqTable.bzseq[producerId] field from node A makes a transition when the busy loop is finally started and packet 21 is retransmitted as packet 24. Node B detects this transaction and now evaluates whether packet 24 can be accepted, based on whether its receive buffer is full. In the assumed example, both retried packets 24 and 25 are accepted.

The basic concept for initiating a busy retry loop in the potential presence of errors is to wait in starting the loop until acknowledgments for all packets prior to the first retried busy packet are received and verified as good. This is necessary since changing the state of the pLabel.bzseq bit causes all node packets to reset their seqTable.busy[] and seqTable.acpt[] fields. These fields had maintained error retry state for generating packet accept/reject and acknowledgments prior to the busy retry loop. These fields are reset only when node A verifies that no error conditions remain
outstanding.

Producer nodes: Request Queue State

In a preferred embodiment,, each producer node may have up to the following three types of outgoing queues in addition to a register, and two types of incoming queues or structures (see Figure 39)
Outgoing:

1. a new request queue of unissued requests, presumably in the transaction layer (above the link layer);
2. a response-pending queue of unissued response packets to other nodes, again in the transaction layer;
3. a "send_pkt" queue of send and response packets in process, which are issued to the

ringlet, in the link layer;

4. a "retry_pkt" queue of busy packets to retry [N.B. this applies only to a busy retry embodiment];
5. a "retry_pkt" register, to both fill the retry_pkt queue and to recycle packets for error retry to the send_pkt queue.

Incoming:

1. a queue of incoming acknowledgments to this node in the link layer;
2. a structure for requests which are awaiting responses, in the transaction layer.

For outgoing send and response packets in the transaction layer, the queues may be in some cases multiple queues, where there is no ordering dependency between the queues. An example is the multiple ports of a switch.

For response packets to this node in SSO ordering mode, responses from a given destination node will always come in order, while no ordering is guaranteed for responses between different nodes. Consequently, the send packet structure awaiting responses is shown as a structure rather than a queue.

Response packet usage in SSO ordering spaces

IEEE P1394.2 specifies two types of operations to write data to a destination node: a responseless move transaction and a write transaction with a response. The SSO ordering structure guarantees ordered delivery of data; what then is the difference between move and write transactions?

Write send packets provide a positive indication, generated by the destination node, that the write data actually reached this node. That is, it positively indicates completion of the operation. Move send packets have no positive indication of completion.

There are two questions that arise from this distinction:

1. What can the positive indication of completion be used for?
2. What architectural extensions can be combined with move transactions to ensure that ordering is maintained in an imperfect link topology, specifically covering invalid addresses and "stuck at" receiver nodes that refuse to accept new packets? The posi-

tive indication of completion for the write transaction will be visible to the transaction layer. Whether it can be visible at some system level above this depends on a transaction level architecture that is beyond the scope of this discussion. The point of SSO ordering is that no structure is necessary to ensure ordering.

Here are two possible uses for write transactions within the transaction layer; this list is hardly exhaustive. First, write transactions can be used as the basis for a reliability architecture. If a send packet failed to receive a response in a specified period of time, a node timeout could be reported. In this usage, there is no a priori requirement for response completion prior to starting the next send packet.

In the second use for write transactions is to use them as a basis for high availability architectures. In this application, the response to the write signals the completion of a "write commit" to stable storage. The problem with this is what to do with the response. Does one use it to gate transmitting the next send packet? If so, this dramatically slows down the packet delivery rate. Does the system architecture support a mechanism to use the response as a completion barrier indicator? Most (including UPA) do not.

Send packet queue structure

Since the address spaces to P1394.2 may, in general, be both ordered and non-ordered, the node must maintain a structure of data for each command to identify and support both addressing models. This proposed structure consists of three fields:

```
Send_pkt_SSO structure, one per send or response packet
send_pkt.val       (1)   /* Entry valid */
send_pkt.sso       (1)   /* From command address space */
send_pkt.done      (1)   /* ACK_done recvd, response may be reqd */
send_pkt.busy      (1)   /* ACK_busy received, awaiting retry */
send_pkt.init_er   (1)   /* Initial (first) packet in error retry loop */
send_pkt.no_xmit   (1)   /* No transmit to ringlet during error loop */
send_pkt.err_ret   (1)   /* Error retry state in error retry loop */
send_pkt.seq       (6)   /* Seq number for this send/response pkt */
send_pkt.packet    (N)   /*N bit send / receive packet */
```

Other control bits may be used for implementation purposes beyond SSO ordering as described here.

Conceptually, send_pkt[] may be viewed as a FIFO queue, with newly sent packets entering at the top and concluded packets coming out the bottom as acknowledgment packets in a corresponding FIFO are received. Any send packets acknowledged as "ack_done" may in turn may be awaiting a response.

In addition to the send_pkt FIFO queue, other queues, registers, muxes, state machines, and control logic are described in the supporting flow diagrams.

Producer node pointers for CRC error loop

Certain producer node variables are needed for CRC error loop and busy retry loop handling. Global variables for CRC error loop operation include these, where the prefix "or_" indicated "ordered".

```
or_last_ct        (6)   /* Sequence count for latest SSO ordered pkt
                           sent to the local ringlet */
no_last_ct        (6)   /* Sequence count for latest non-SSO ordered
                           pkt sent to the local ringlet */
or_rcvd_ct        (6)   /* Sequence count for latest SSO ordered pkt
                            received from the local ringlet */
```

When a packet is sent to the local ringlet, its pLabel field is set to correspond with send_pkt fields, which are carried in the packet path into the send_pkt register. In general:

```
pLabel.sso      = send_pkt_reg.sso
pLabel.bzseq    = bzseq /* Global producer node state bit */
pLabel.seq      = send_pkt_reg.seq
```

What value is assigned to pLabel.seq depends on the operation (e.g., new packet, busy retry packet, or CRC retry); see details below.

Receiver pointer comparison: supporting a wrapped modulo 64 count

From the "Proposal overview" section above, it was noted that three comparisons can be important to the receiver operation:

1. The receiver node can detect a valid new, SSO ordered packet from a given producerId

if:

```
pLabel.seq = seqTable.seq[producerId] + 1, and
pLabel.seq > seqTable.acpt[producerId]
```

2. The seqTable.seq[producerId] field is reset to the current pLabel.seq value when this node detects that this producerId is executing a CRC error loop. This is indicated by comparing the seq fields for a packet - or acknowledge - from this node:

```
if {pLabel.seq <= seqTable.seq[producerId]} then
    {seqTable.seq[producerId] = pLabel.seq
    }
```

3. Finally, each node can detect a packet with a sequence error from the producerId if:

pLabel.seq > seqTable.seq[producerId] + 1

Supporting both "greater than" and "less than or equal" comparisons with a modulo 64 counter can be achieved if the producerId ensures that the comparison will never exceed a difference at the receiver node of 31 and the count is defined as below:

"A <= B" if:

- high order bit equal and A <= B with lower order bits
- high order bit unequal and A > B with remaining bits

"A > B" if:

- high order bit equal and A > B with lower order bits
- high order bit unequal and A <= B with remaining bits

Special Issues

This section provides additional explanation and emphasis on some subtle details with possible implementations of the invention.

The first of these problems dealt with reliably resetting appropriate seqTable[producerId] fields following a CRC error. Once a CRC error has occurred, the fields in the seqTable.seq[producerID] for every node in the ringlet may contain values which are not in synch with other nodes within the ringlet.

At the same time, since the seqTable[] values are **only** updated when a valid, correctly sequenced packet is identified at the receiver node, then the values must be considered as

correct insofar as this node is concerned. They may, however, be out of synch with respect to both the sending node and other ringlet nodes.

So the task of the producerId node is to reliably get all nodes back into synch. The problem is in achieving this reset: what if the first packet in the error loop itself has a CRC error?

A simple solution to this problem is for the producerId node performing the error handling loop to begin its error loop by transmitting - in non-pipelined mode - the last known good, completed packet. The producerID node would retry this packet until it received a valid acknowledgment. Once this is received, then every node on the ring must have its seqTable.seq[] field for this producerID set to the same value. The same argument holds for resetting the seqTable.bzseq[] field. But the seqTable.busy[] field reflects the state of the acknowledgment to the last valid, completed packet delivered from this producerId to this receiving node. In considering how to handle the next valid packet (any retried packet will be rejected), this field has validity. This field, consequently, must not be reset in the reset interval. Similarly, the seqTable.acpt[] field is also left unmodified.

Note that by retrying a known good, completed packet (which would be acknowledged as valid and rejected by the target node in the ringlet) the producerID node is transmitting a packet for only its the packet sequence number, rather than its data content.

The same approach of resending a known packet until a valid ACK is returned, can be used for ringlet initialization to initialize the above mentioned fields. But other fields, such as seqTable.busy[] and seqTable.acpt[], must be reset by other means, such as using a non-SSO space to write into the seqTable.acpt[] field the value (pLabel.seq).

The second problem with the original proposal dealt with the question of ambiguity in the producerID node determining whether packets in its error loop had been completed or not. That is, if any (send/receive) and acknowledge pair is found to have a CRC error, then the producerID node cannot determine whether the CRC error occurred before the packet reached its ringlet destination, or after. So it cannot reliably mark a retried packet as "completed".

The solution to this problem was to add the (6 bit) seqTable.acpt[] field to each receiver node's seqTable array to record the last valid sequence number detected at this node prior to having its seqTable.seq[] reset by a packet with a lower value pLabel.seq from this producerId, or the sequence number of the first packet rejected for busy retry (indicated by the seqTable.busy[] field. Note that tracking the seqTable.acpt[] field means that:

1. The last accepted packet at this node must have a sequence number <= seqTable.acpt[], so any new, unseen packet must have a sequence value > seqTable.acpt[], provided that seqTable.busy[] is reset.
2. The seqTable.acpt, in the absence of errors or busy packets, tracks the current pLabel.seq value. This fact is crucial in creating a modulo 64 rolling count definition.

Notes regarding the Example of Figures 39-60

Figures 39-60 show the state of the system of the invention at successive stages of processing packets wherein a busy condition has arisen with respect to a consumer node, a CRC (or other) error condition has also arisen, and the "busy" node is not well-behaved -- i.e., it has been persistently busy for longer than some predetermined length of time or number of packets sent or received, or it is erroneously transmitting ack_busy responses.

In the example of Figures 39-60, it will be noted that the or_last_ct is the count number of the last ordered packet sent, and that or_rcvd_ct refers to the count of the last ordered packet received. The or_rcvd_ct value is frozen if a busy retry is required (which is detected in the resend logic).

For this example, it is assumed that there are previous busy packets, e.g. 54 and 59, which are in the retry_pkt queue. Note that:

(or_last_ct - or_rcvd_ct) = (23-59) modulo 64

= 28

and 28 is less than or equal to the threshold for the difference (or_last_ct - or_rcvd_ct).

What is claimed is:

1. A system for maintaining a sequence of packets transmitted by at least one producer node to at least one consumer node in a computer network supporting , including:

a first send subsystem of said producer node configured to maintain sequence and packet state information for at least one said packet sent by said producer node; and

a first receive subsystem of said producer node configured to maintain sequence and packet state information for an acknowledgment sent by said consumer node to said producer node, and to detect a condition wherein a sent packet has not resulted in an acknowledgment

a second receive subsystem configured of said producer node configured to detect busy acknowledgments from said consumer node;

a second send subsystem of said consumer node configured to maintain sequence and packet state information for said acknowledgment;

a second receive subsystem of said consumer node configured to maintain sequence and packet state information for said packet sent by said producer node, and to maintain overall sequence state information and packet accept state information for all packets received by said consumer node from said producer node, and further configured to reject packets when at least one said busy acknowledgment is detected; and

a node monitoring subsystem in said producer node, configured to determine when at least one said consumer node in said network fails to meet a predetermined response criterion.

2. The system of claim 1, wherein said node monitoring subsystem includes a busy acknowledgment queue configured to store a plurality of busy acknowledgments received from said consumer node.

3. The system of claim 2, wherein said node monitoring subsystem is configured to make said determination based upon said busy acknowledgment queue storing at least a predetermined number of said busy acknowledgments.

4. The system of claim 1, wherein said predetermined response criterion includes a predeter-

mined amount of time during which said producer node receives a plurality of busy acknowledgments from said consumer node.

5. The system of claim 1, wherein said predetermined response criterion includes a predetermined amount of time during which said producer node receives no acknowledgment from said consumer node.

6. The system of claim 1, wherein said system is configured to process at least a first packet comprising a nonidempotent request while maintaining unaltered a state of a node to which said ~~first packet is a~~ddressed, upon receiving said first packet more than once.

7. The system of claim 1, including a retry subsystem configured to retry at least one packet including a nonidempotent request while maintaining unaltered a state of said consumer node upon receiving a second instance of said retried packet.

8. The system of claim 1, wherein said network is a ringlet ~~network~~.

9. The system of claim 1, wherein said network includes at least one ordered node configured to maintain strong sequential ordering and at least one unordered node configured in a manner other than to maintain strong sequential ordering.

10. The system of claim 1, wherein said network includes at least a first node configured to support nonidempotent commands and second node configured not to support nonidempotent commands.

11. The system of claim 1, wherein said network includes at least one dynamic node configurable, at different times, both in a manner to maintain both strong sequential ordering and in a manner other than to maintain strong sequential ordering.

12. The system of claim 1, further including:
an error detection subsystem; and

a reset subsystem configured, upon detection of an error by said error detection subsystem upon sending of a current packet, to reset said sequence of each said consumer node to a common network value.

13. The system of claim 12, wherein said reset subsystem is configured to cause said producer node to resend a previously sent packet at least once, and repeatedly as required, until a valid acknowledge packet is received by said producer node.

14. The system of claim 13, wherein said previously sent packet is a packet for which said producer node has received an acknowledge done packet.

15. The system of claim 14, further including a first retry subsystem configured to send retry packets between said previously sent packet and said current packet, inclusive.

16. The system of claim 15, further including a retry packet validity check subsystem configured to determine validity of retry packet acknowledgments received at said producer node in response to sending said retry packets.

17. The system of claim 16, further including a second retry subsystem configured to send said retry packets when said retry packet validity check subsystem determines that a retry packet acknowledgment has an invalid status.

18. The system of claim 13, wherein said reset subsystem is configured to remove a data field from said previously sent packet before resending.

19. The system of claim 13, wherein said reset subsystem includes:

a sequence validity state subsystem configured to maintain a sequence validity state value at each said consumer node, and to reset each said sequence validity state value to a common value upon receipt at each said consumer node of said previously sent packet.

20. The system of claim 15, wherein said retry subsystem includes:

an acceptance validity state subsystem configured to maintain an acceptance validity state value at each said consumer node;

an acceptance validity comparison subsystem configured to generate a comparison of an acceptance validity field of said retry packets with said acceptance validity state value at each said consumer node; and

a retry packet reject subsystem configured to reject each said retry packet whose said comparison meets a predetermined criterion.

21. The system of claim 1, including a plurality of send and consumer nodes, wherein each said consumer node includes one said receive subsystem configured to read state information of a plurality of said packets and said acknowledgments, respectively, even when said packets and acknowledgments are addressed to nodes other than the consumer node reading said state information, to maintain common sequencing information among all nodes that support strong sequence ordering.

22. The system of claim 1, including a plurality of said producer nodes and a plurality of said consumer nodes, wherein:

at least a subset of said producer and consumer nodes are configured as strong sequential ordering (SSO) nodes to send and receive, respectively, SSO packets, and to receive and send, respectively, SSO acknowledgments of said SSO packets; and

each said producer and consumer node of said subset is configured to read said SSO packets and/or said SSO acknowledgments.

23. The system of claim 1, configured to maintain unaltered an order of request packets as transmitted by producer nodes on said network, and to maintain unaltered an order of response packets generated in response to said request packets by consumer nodes on said network.

24. A system for maintaining a sequence of packets transmitted by at least one producer node to at least one consumer node in a computer network, including:

a sequence check subsystem in said consumer node configured to determine whether at least one packet in the sequence is in a valid position in the sequence; and

a busy acknowledgment detection subsystem in said producer node configured to determine whether a received acknowledgment indicates a busy condition in said consumer node; and

a node monitoring subsystem in said producer node, configured to determine when at least one said consumer node in said network fails to meet a predetermined response criterion.

25. The system of claim 24, wherein said node monitoring subsystem includes a busy acknowledgment queue configured to store a plurality of busy acknowledgments received from said consumer node.

26. The system of claim 25, wherein said node monitoring subsystem is configured to make said determination based upon said busy acknowledgment queue storing at least a predetermined number of said busy acknowledgments.

27. The system of claim 24, wherein said predetermined response criterion includes a predetermined amount of time during which said producer node receives a plurality of busy acknowledgments from said consumer node.

28. The system of claim 24, wherein said predetermined response criterion includes a predetermined amount of time during which said producer node receives no acknowledgment from said consumer node.

29. A system for maintaining a sequence of packets transmitted by at least one producer node to at least one consumer node in a computer network, including:

a pipelining subsystem configured to pipeline packets from said producer node in the absence of acknowledgment to said producer node of previously transmitted, unprocessed packets;

a counter subsystem configured to determine a sequence count difference between a first said unprocessed packet and a current packet;

a cutoff subsystem configured to terminate said pipelining of said packets when said sequence count difference reaches a predetermined threshold;

a busy detection subsystem configured to detect a busy condition at said consumer node;

a packet suspension subsystem configured to suspend pipelining of said packets when said busy condition is detected; and

a node monitoring subsystem in said producer node, configured to determine when at least one said consumer node in said network fails to meet a predetermined response criterion.

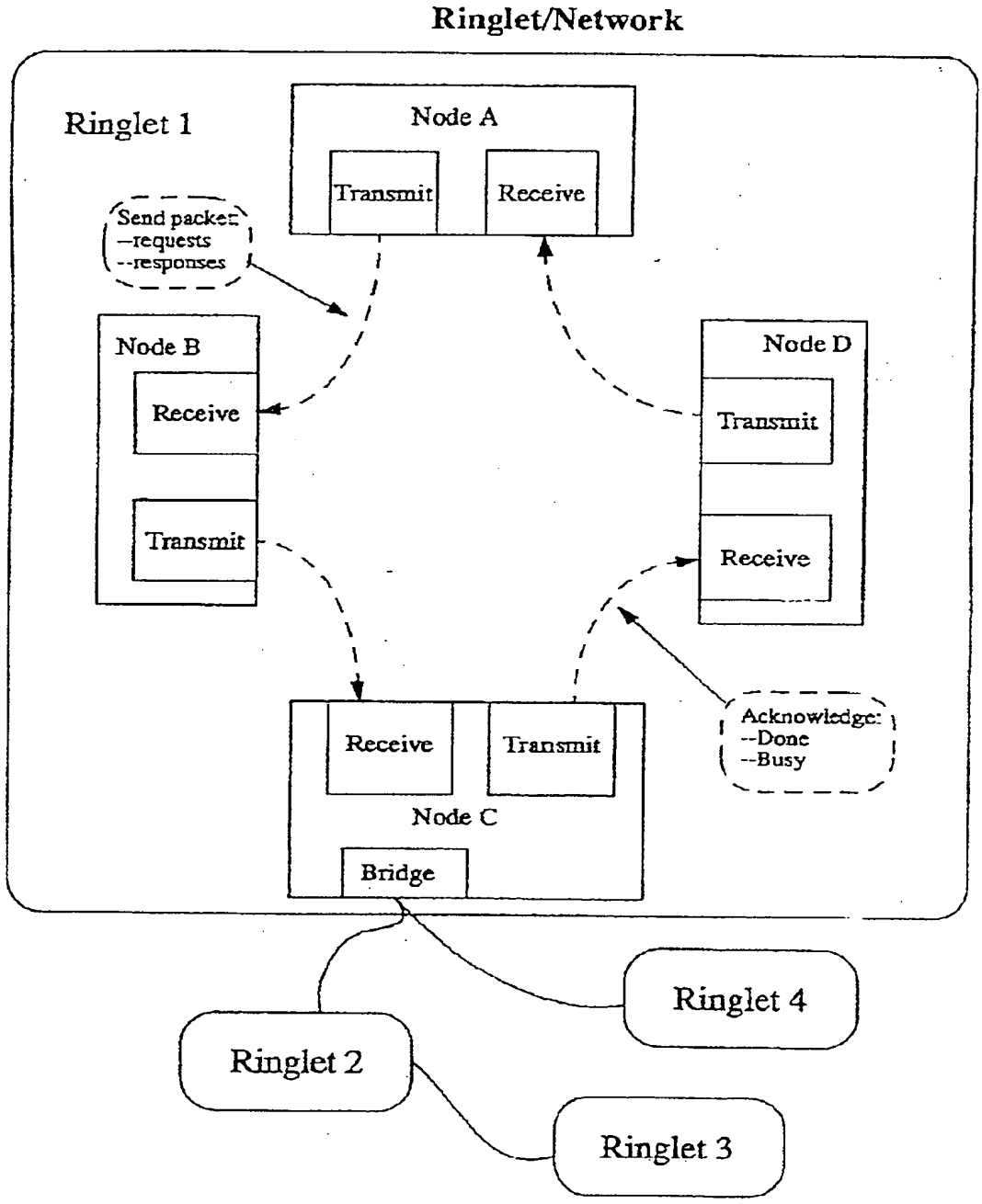
30. The system of claim 29, wherein said node monitoring subsystem includes a busy acknowledgment queue configured to store a plurality of busy acknowledgments received from said consumer node.

31. The system of claim 30, wherein said node monitoring subsystem is configured to make said determination based upon said busy acknowledgment queue storing at least a predetermined number of said busy acknowledgments.

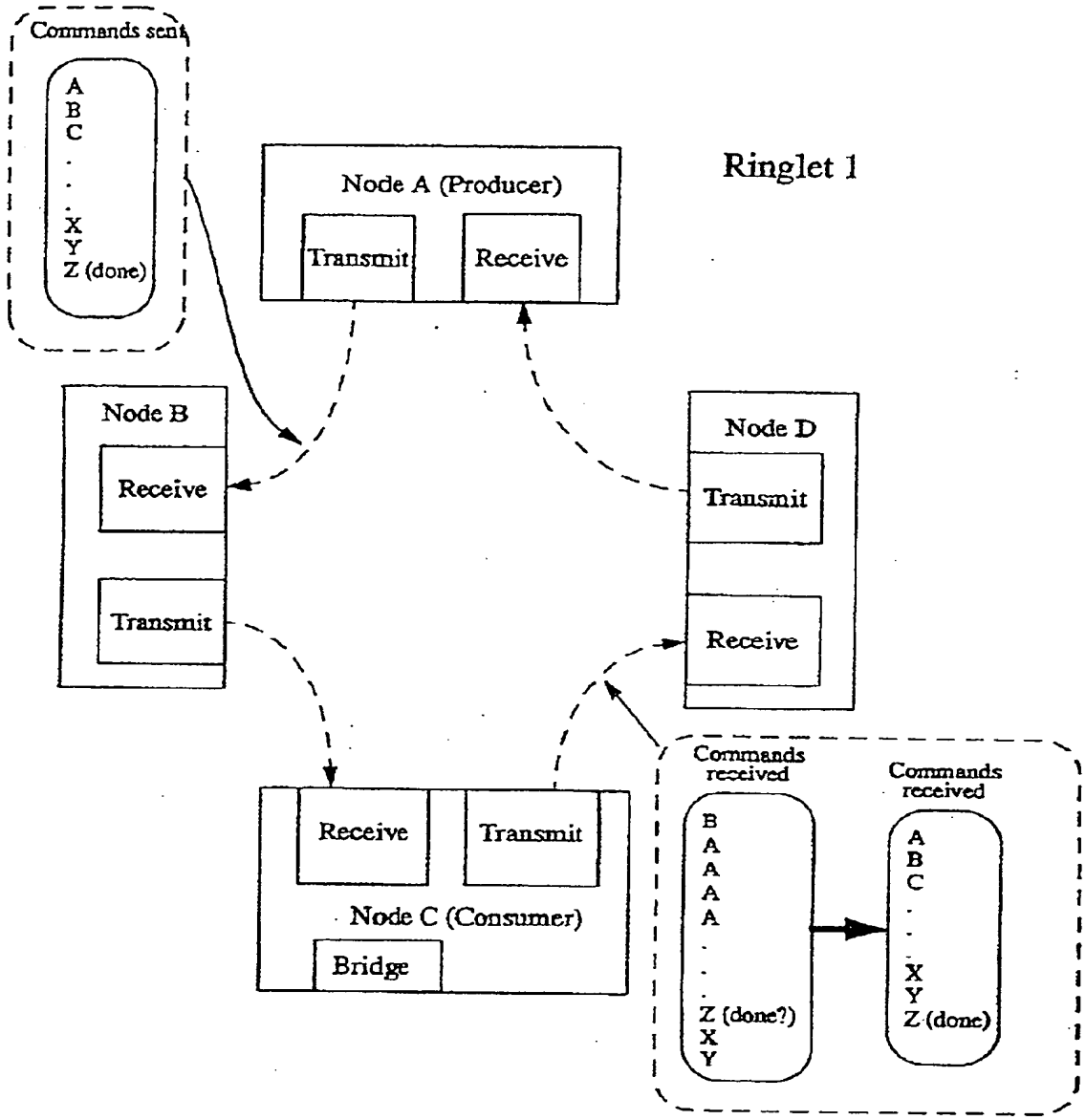
32. The system of claim 29, wherein said predetermined response criterion includes a predetermined amount of time during which said producer node receives a plurality of busy acknowledgments from said consumer node.

33. The system of claim 29, wherein said predetermined response criterion includes a predetermined amount of time during which said producer node receives no acknowledgment from said consumer node.

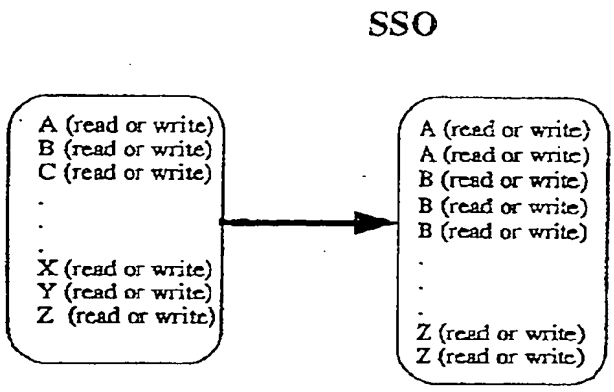
〔Ｆｉｇ．１〕


Ringlet/Network
Ringlet 1
Node A
Transmit
Receive
Send packet
--requests
--responses
Node B
Receive
Transmit
Node D
Transmit
Receive
Receive
Transmit
Node C
Bridge
Acknowledge:
--Done
--Busy
Ringlet 4
Ringlet 2
Ringlet 3

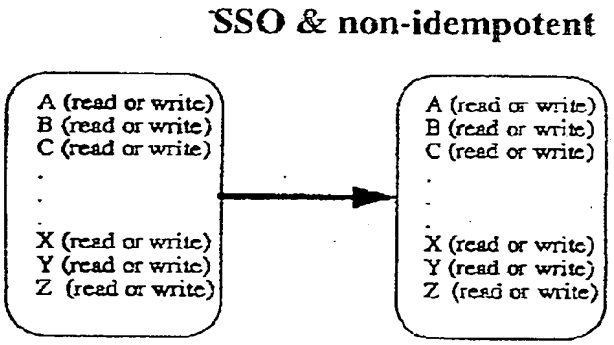
〔Ｆｉｇ．２〕


Commands sent
A
B
C
X
Y
Z (done)
Node A (Producer)
Transmit
Receive
Ringlet 1
Node B
Receive
Transmit
Node D
Transmit
Receive
Receive
Transmit
Node C (Consumer)
Bridge
Commands received
B
A
A
A
A
Z (done?)
X
Y
Commands received
A
B
C
X
Y
Z (done)

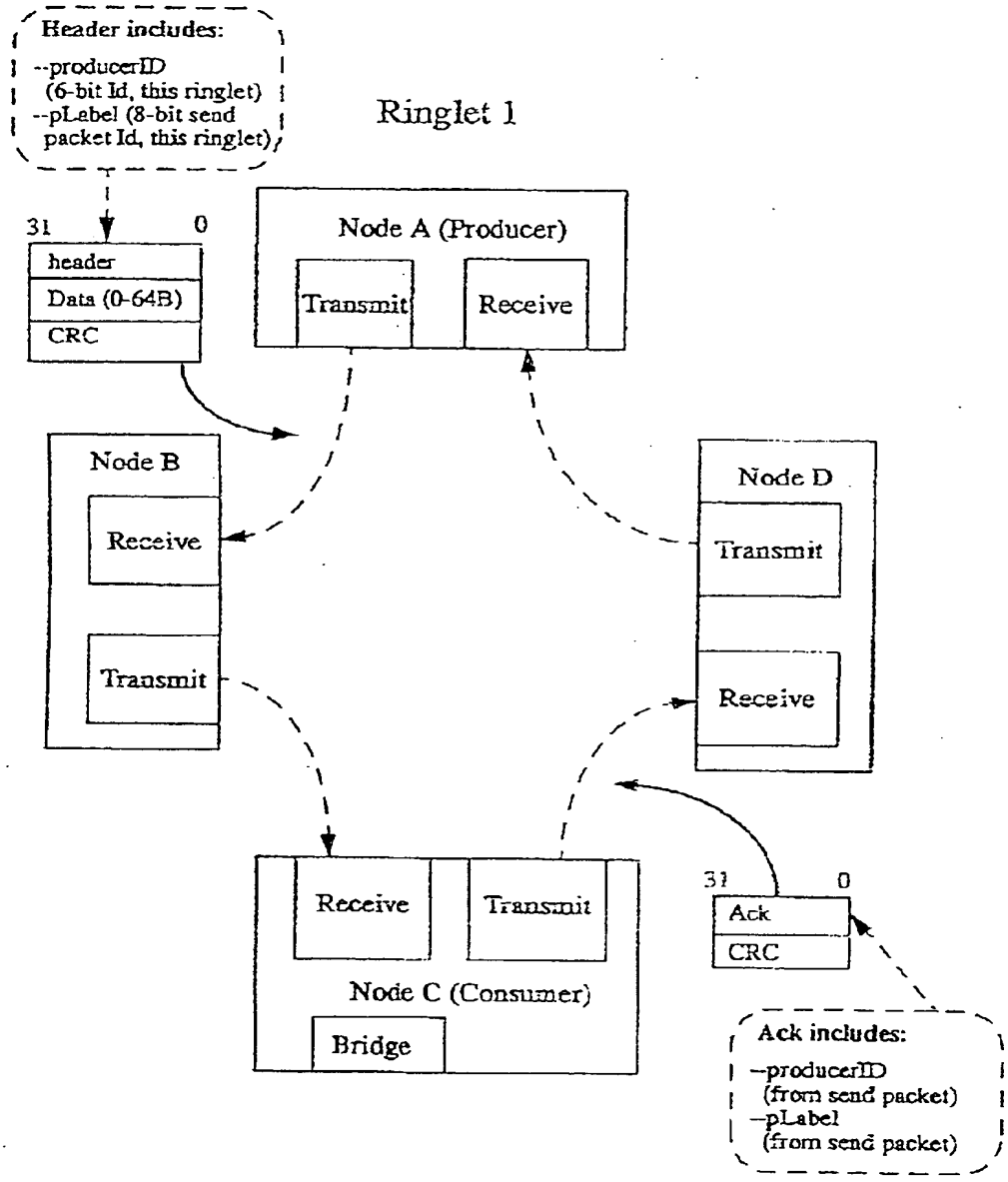
【Ｆｉｇ．３】


SSO
A (read or write)
B (read or write)
C (read or write)
X (read or write)
Y (read or write)
Z (read or write)
A (read or write)
A (read or write)
B (read or write)
B (read or write)
B (read or write)
Z (read or write)
Z (read or write)

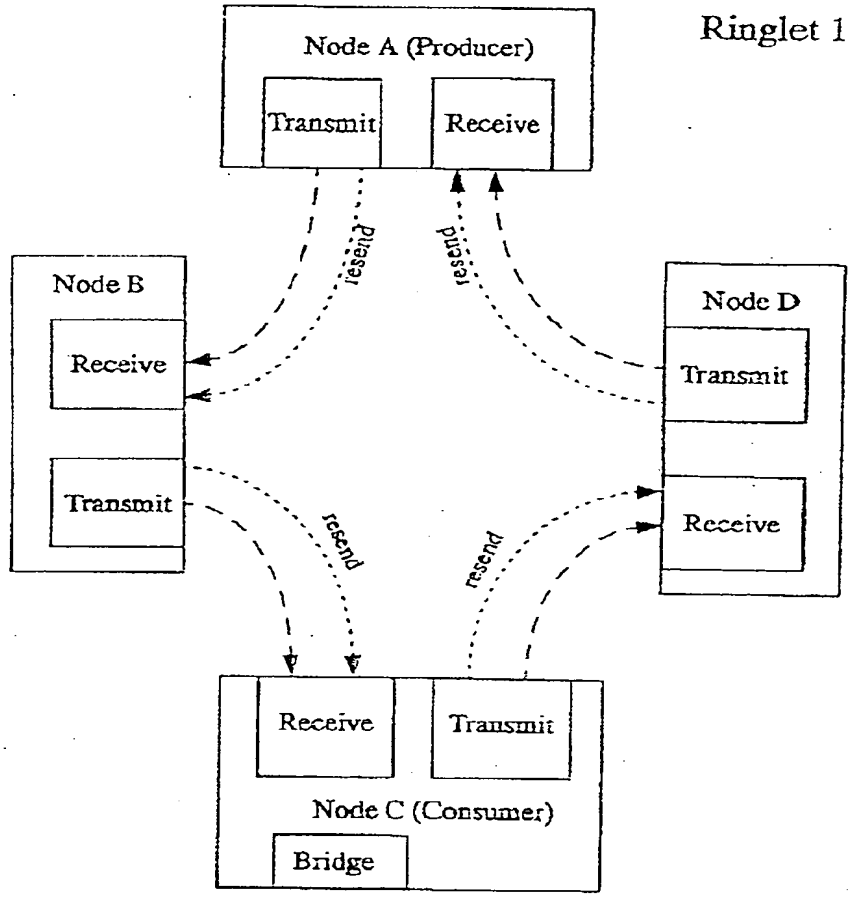
【Ｆｉｇ．４】


SSO & non-idempotent
A (read or write)
B (read or write)
C (read or write)
X (read or write)
Y (read or write)
Z (read or write)
A (read or write)
B (read or write)
C (read or write)
X (read or write)
Y (read or write)
Z (read or write)

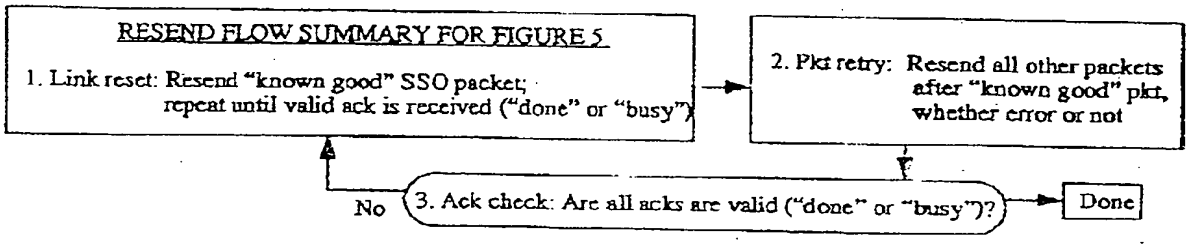
【Ｆｉｇ．５】


Header includes:
--producerID
(6-bit Id, this ringlet)
--pLabel (8-bit send
packet Id, this ringlet)
Ringlet 1
31
0
header
Data (0-64B)
CRC
Node A (Producer)
Transmit
Receive
Node B
Receive
Transmit
Node D
Transmit
Receive
Receive
Transmit
Node C (Consumer)
Bridge
31
0
Ack
CRC
Ack includes:
--producerID
(from send packet)
--pLabel
(from send packet)

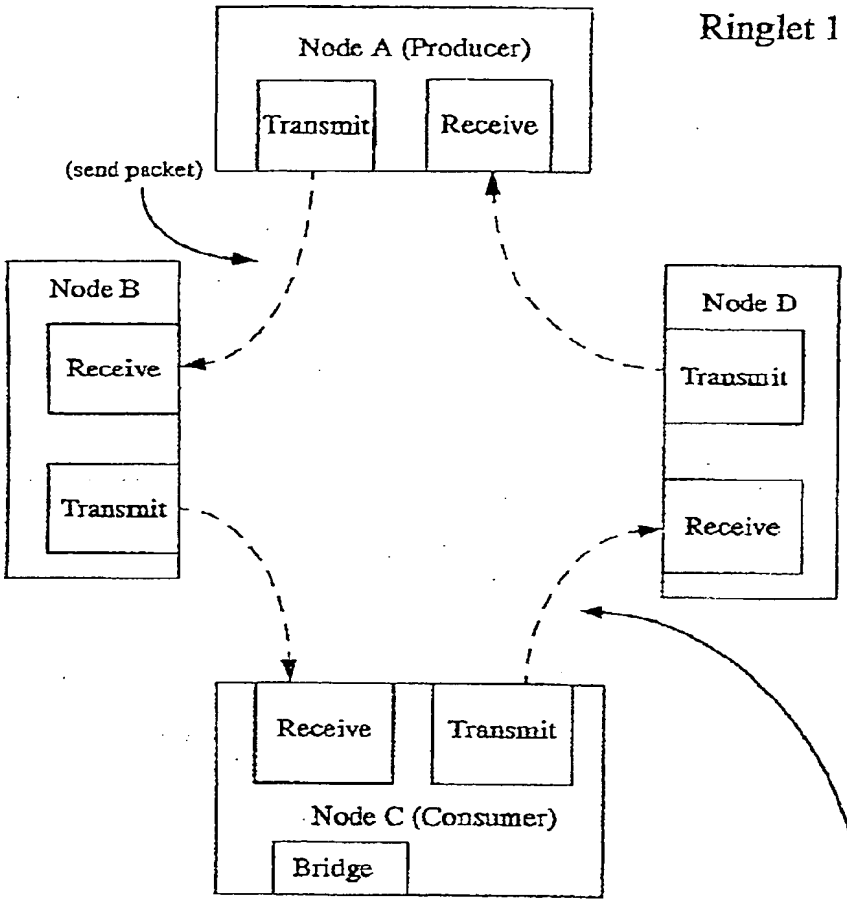
【Ｆｉｇ．６】


Ringlet 1
Node A (Producer)
Transmit
Receive
resend
resend
Node B
Receive
Transmit
Node D
Transmit
Receive
resend
resend
Receive
Transmit
Node C (Consumer)
Bridge

【Ｆｉｇ．６Ａ】

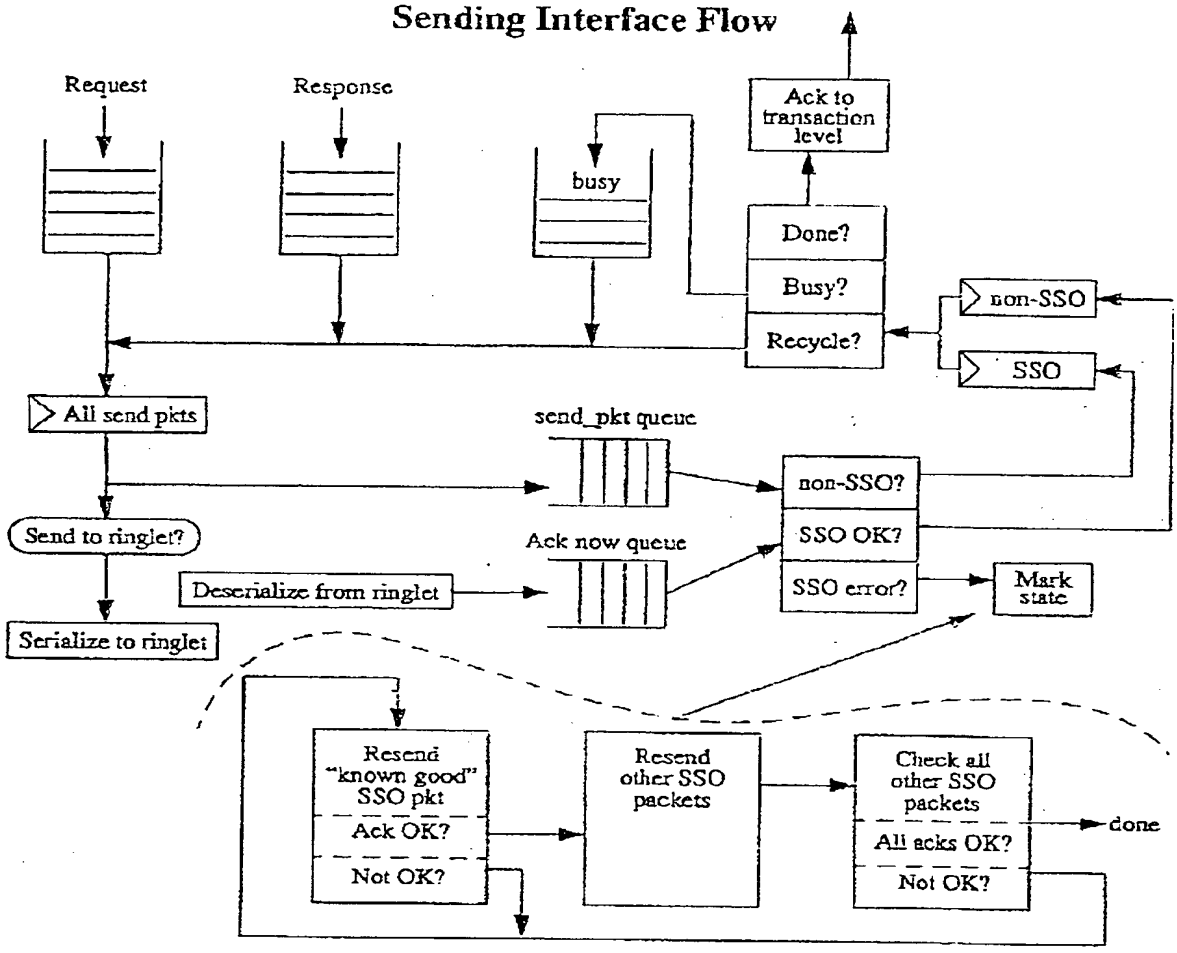

RESEND FLOW SUMMARY FOR FIGURE 5
1. Link reset: Resend "known good" SSO packet;
repeat until valid ack is received ("done" or "busy")
2. Pkt retry: Resend all other packets
after "known good" pkt,
whether error or not
No
3. Ack check: Are all acks are valid ("done" or "busy")?
Done

【Ｆｉｇ．７】



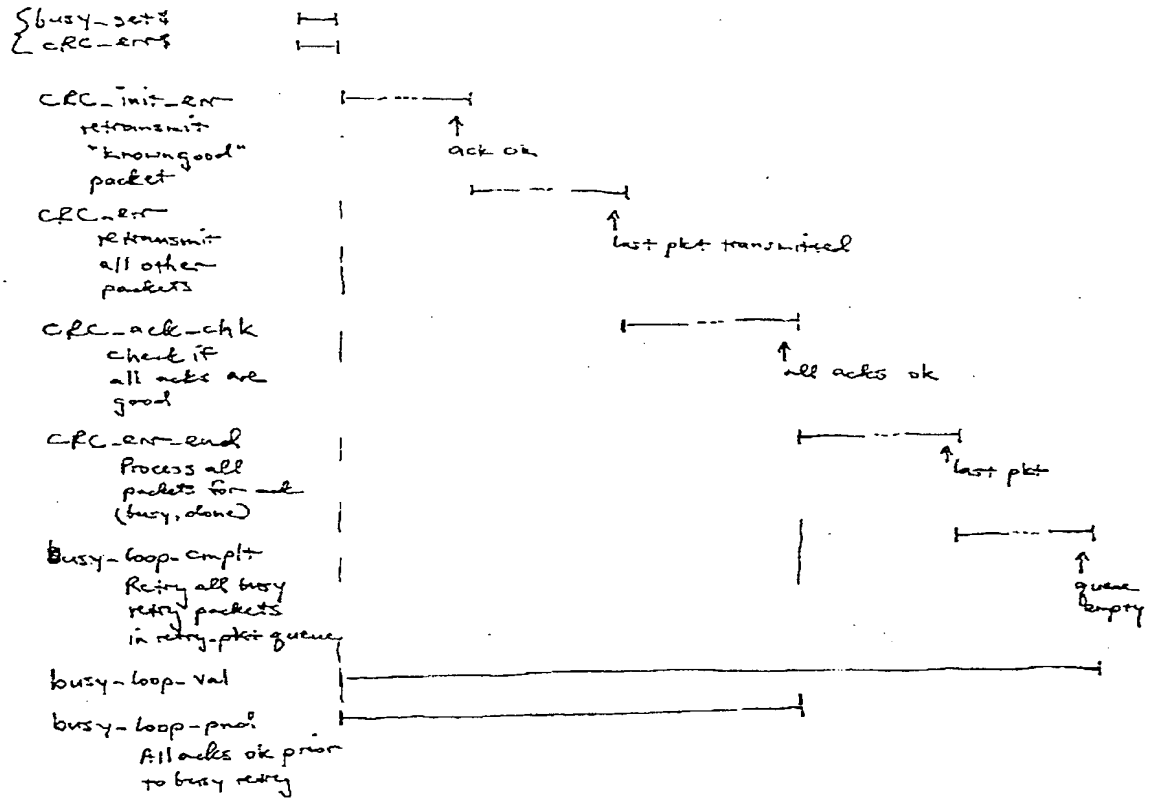
Consumer Action	New Packet Acknowledgment	Resent Packet Acknowledgment
Accept packet	Ack done	—
Reject packet	Ack busy	Ack done Ack busy

【Ｆｉｇ．８】


Sending Interface Flow
Request
Response
busy
Ack to transaction level
Done?
Busy?
Recycle?
non-SSO
SSO
All send pkts
send_pkt queue
non-SSO?
SSO OK?
SSO error?
Mark state
Send to ringlet?
Ack now queue
Deserialize from ringlet
Serialize to ringlet
Resend "known good" SSO pkt
Ack OK?
Not OK?
Resend other SSO packets
Check all other SSO packets
All acks OK?
Not OK?
done

【Fig. 8A】

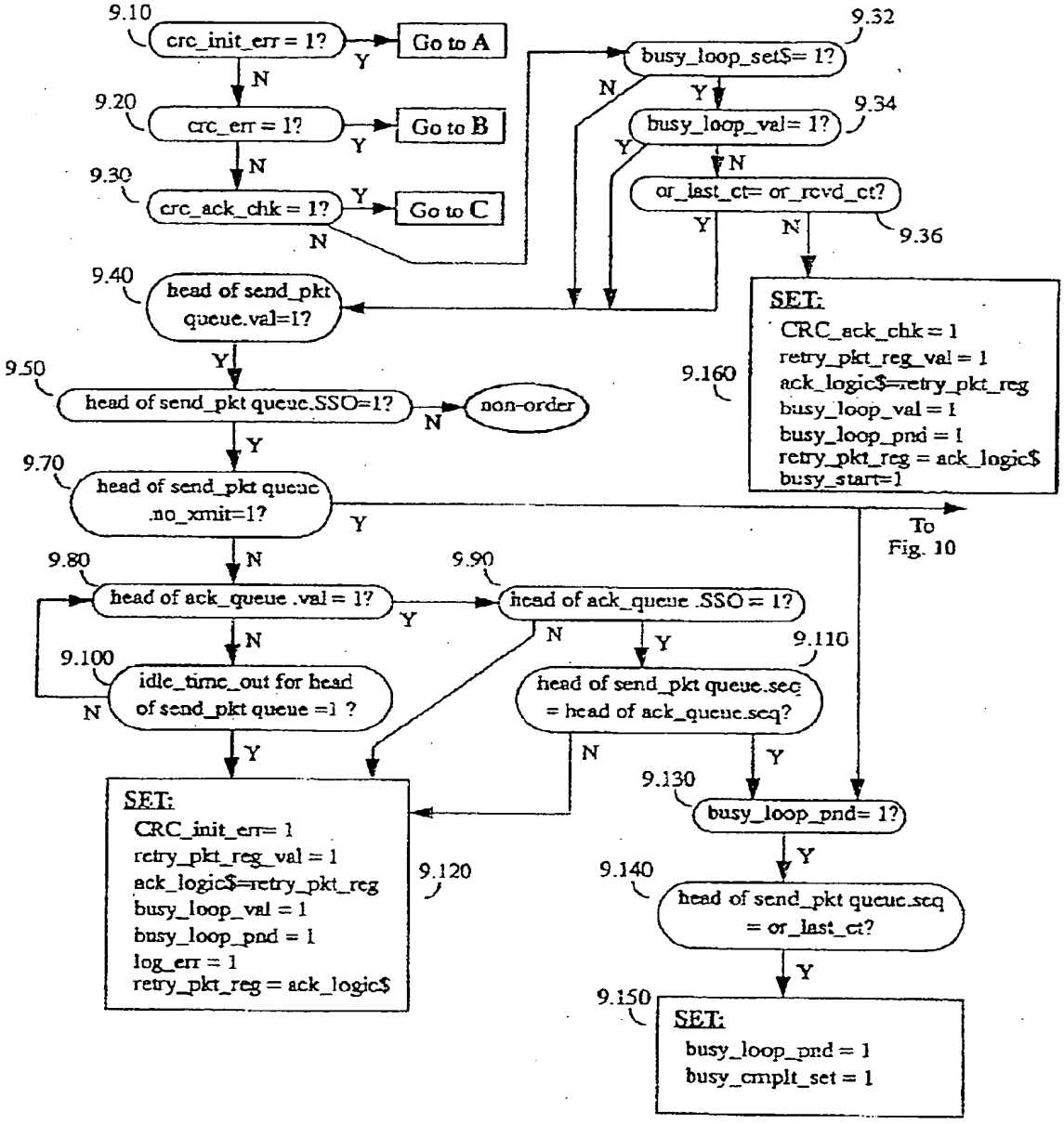
Busy loop timing diagram - user pov



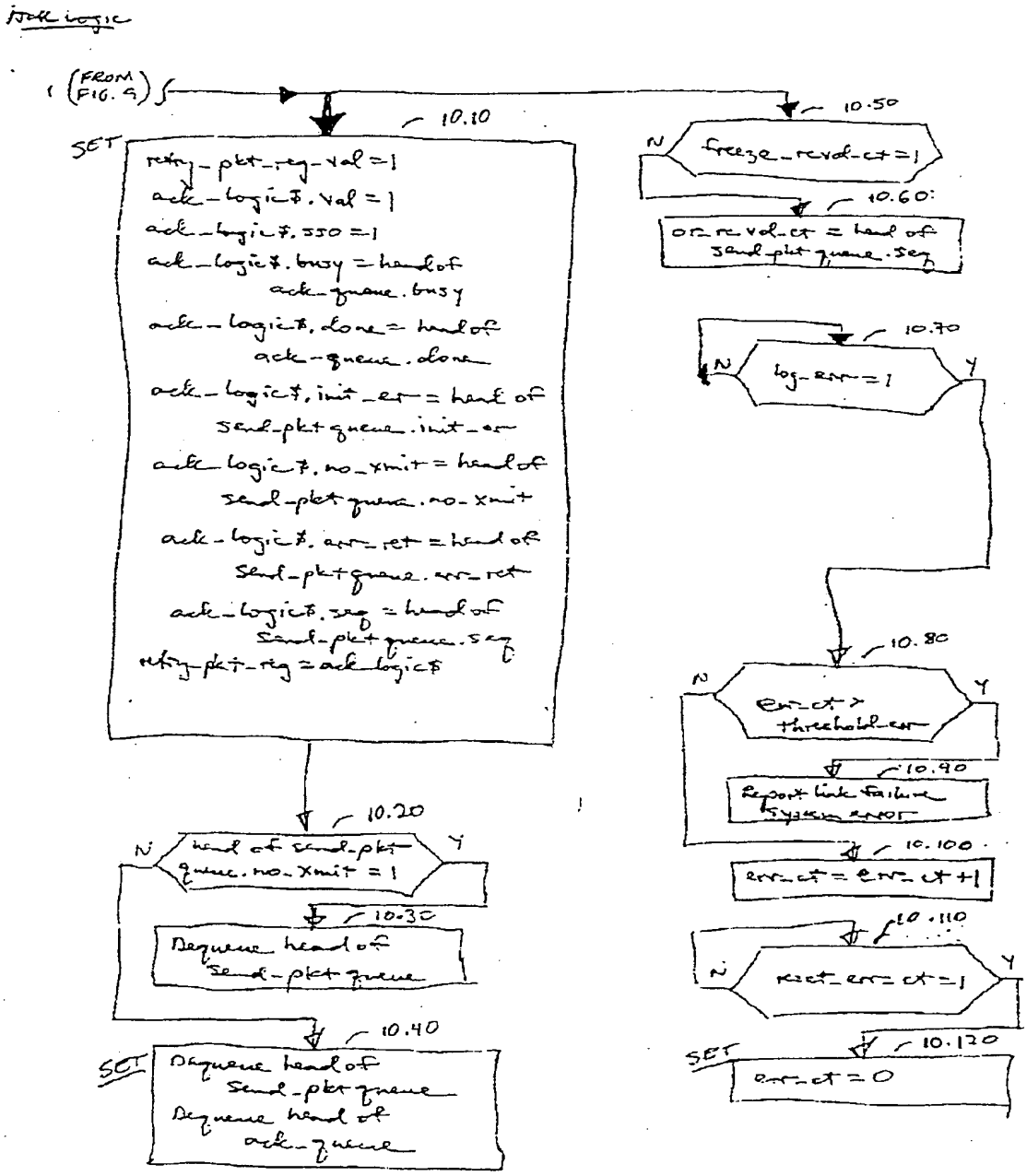
Error loop packet state assignment

	crc_init_err	crc_err	crc_ack_chk
First packet "known good"	init_err	init_err no_xmit	init_err no_xmit
Middle packets	no_xmit	err_ret	no_xmit
Last packet	no_xmit	err_ret	err_ret no_xmit

【Ｆｉｇ．９】

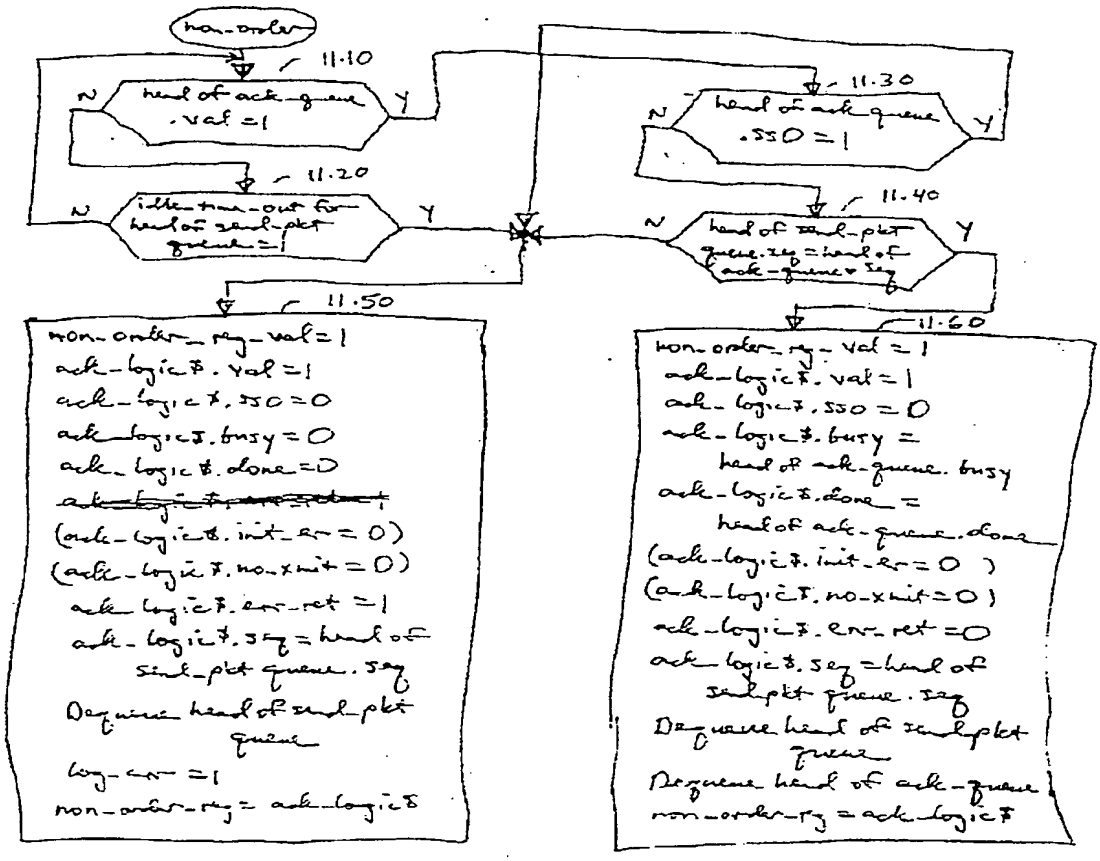

9.10
crc_init_err = 1?
Y
Go to A
N
9.20
crc_err = 1?
Y
Go to B
N
9.30
crc_ack_chk = 1?
Y
Go to C
N
9.32
busy_loop_set$= 1?
N
Y
9.34
busy_loop_val= 1?
Y
N
or_last_ct= or_rcvd_ct?
Y
N
9.36
9.40
head of send_pkt queue.val=1?
Y
9.50
head of send_pkt queue.SSO=1?
N
non-order
Y
9.160
SET:
CRC_ack_chk = 1
retry_pkt_reg_val = 1
ack_logic$=retry_pkt_reg
busy_loop_val = 1
busy_loop_pnd = 1
retry_pkt_reg = ack_logic$
busy_start=1
9.70
head of send_pkt queue .no_xmit=1?
Y
To Fig. 10
N
9.80
head of ack_queue .val = 1?
Y
9.90
head of ack_queue .SSO = 1?
N
Y
9.110
N
9.100
idle_time_out for head of send_pkt queue =1 ?
N
Y
head of send_pkt queue.seq = head of ack_queue.seq?
N
Y
9.130
busy_loop_pnd= 1?
Y
9.140
head of send_pkt queue.seq = or_last_ct?
Y
9.150
SET:
busy_loop_pnd = 1
busy_cmplt_set = 1
SET:
CRC_init_err= 1
retry_pkt_reg_val = 1
ack_logic$=retry_pkt_reg
busy_loop_val = 1
busy_loop_pnd = 1
log_err = 1
retry_pkt_reg = ack_logic$
9.120

【Ｆｉｇ．１０】

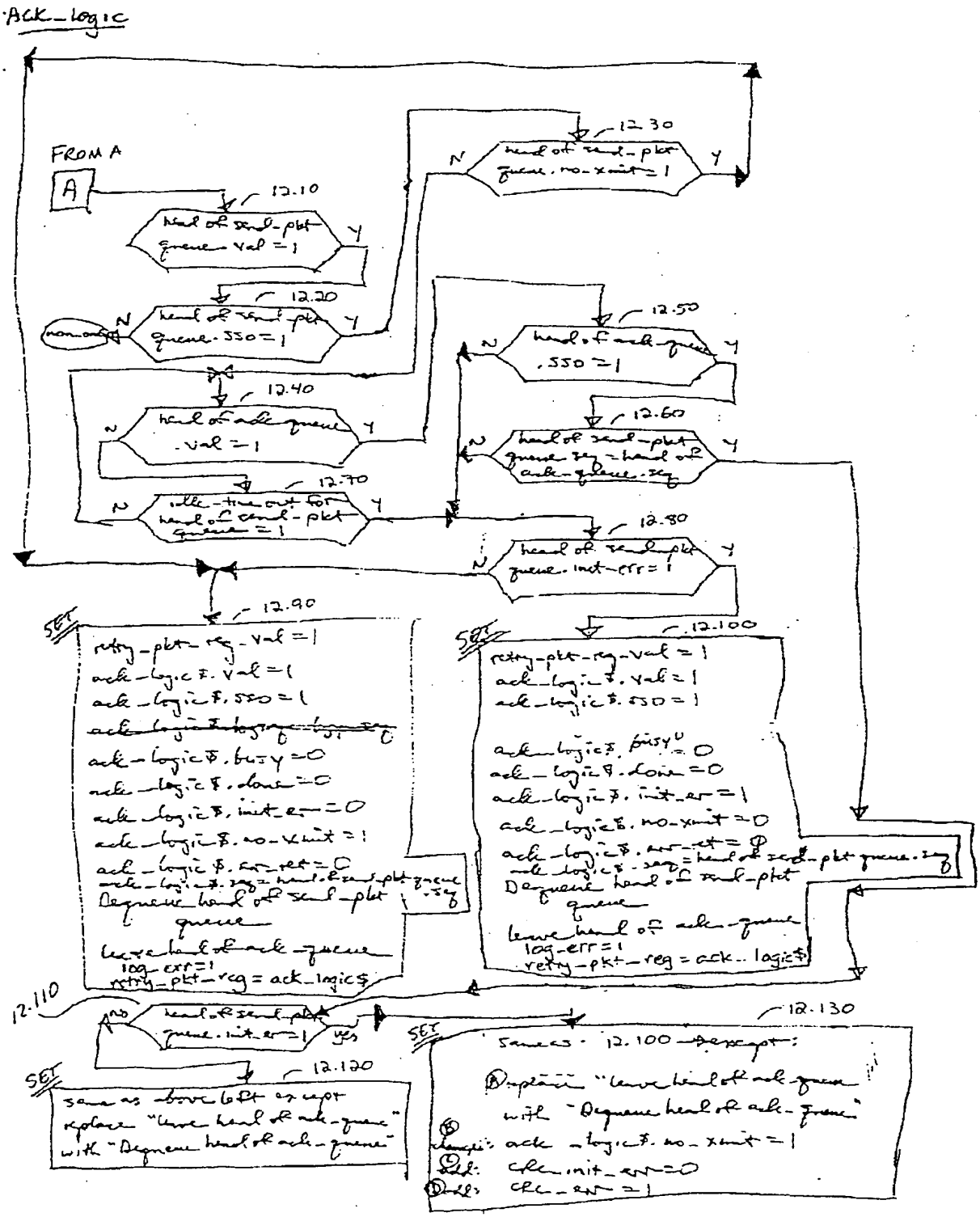

ack logic
(FROM FIG. 9)
10.10
SET
retry_pkt_req.val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = head of ack_queue.busy
ack_logic$.done = head of ack_queue.done
ack_logic$.init_err = head of send_pkt_queue.init_err
ack_logic$.no_xmit = head of send_pkt_queue.no_xmit
ack_logic$.err_ret = head of send_pkt_queue.err_ret
ack_logic$.seq = head of send_pkt_queue.seq
retry_pkt_req = ack_logic$
10.20
N
head of send_pkt_queue.no_xmit = 1
Y
10.30
Dequeue head of send_pkt_queue
10.40
SET
Dequeue head of send_pkt_queue
Dequeue head of ack_queue
10.50
N
freeze_rcvd_ct = 1
10.60
err_rcvd_ct = head of send_pkt_queue.seq
10.70
N
log_err = 1
Y
10.80
N
err_ct > threshold_err
Y
10.90
Report link failure System error
10.100
err_ct = err_ct + 1
10.110
N
reset_err_ct = 1
Y
10.120
SET
err_ct = 0

[Fig. 11]

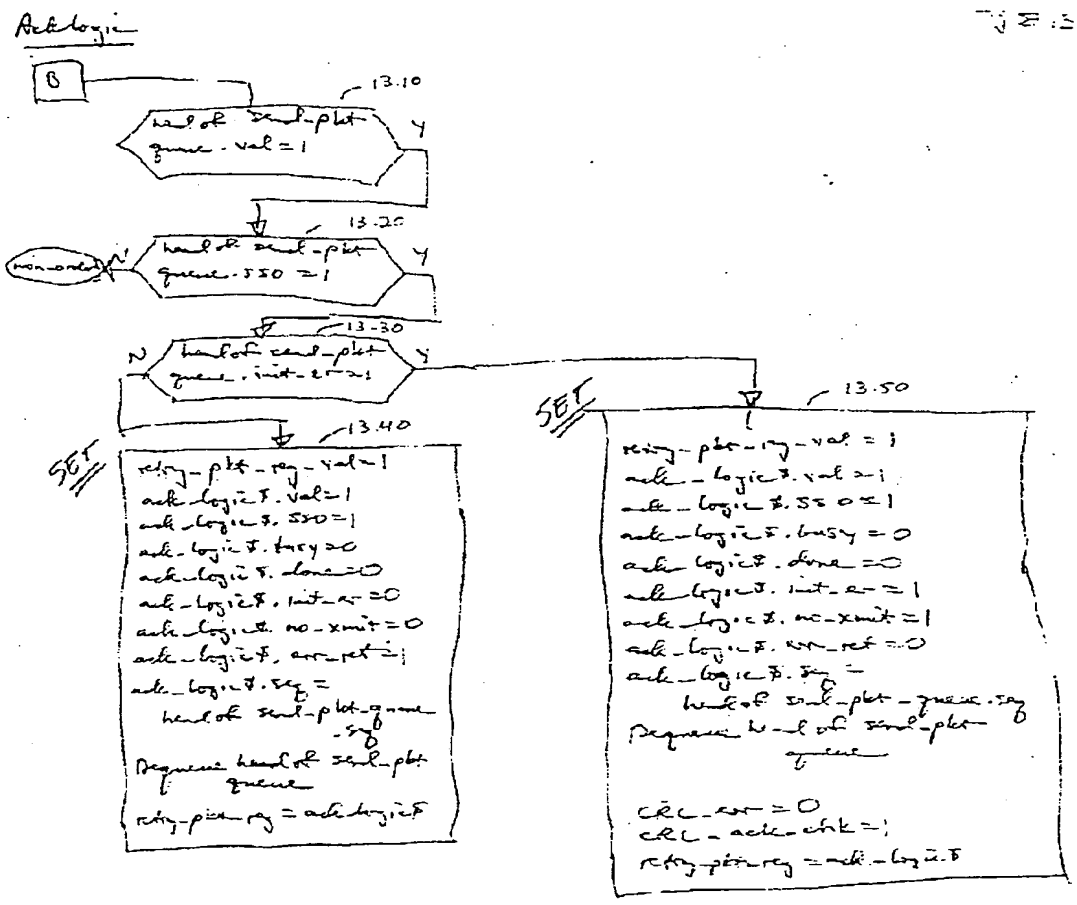
Ack logic



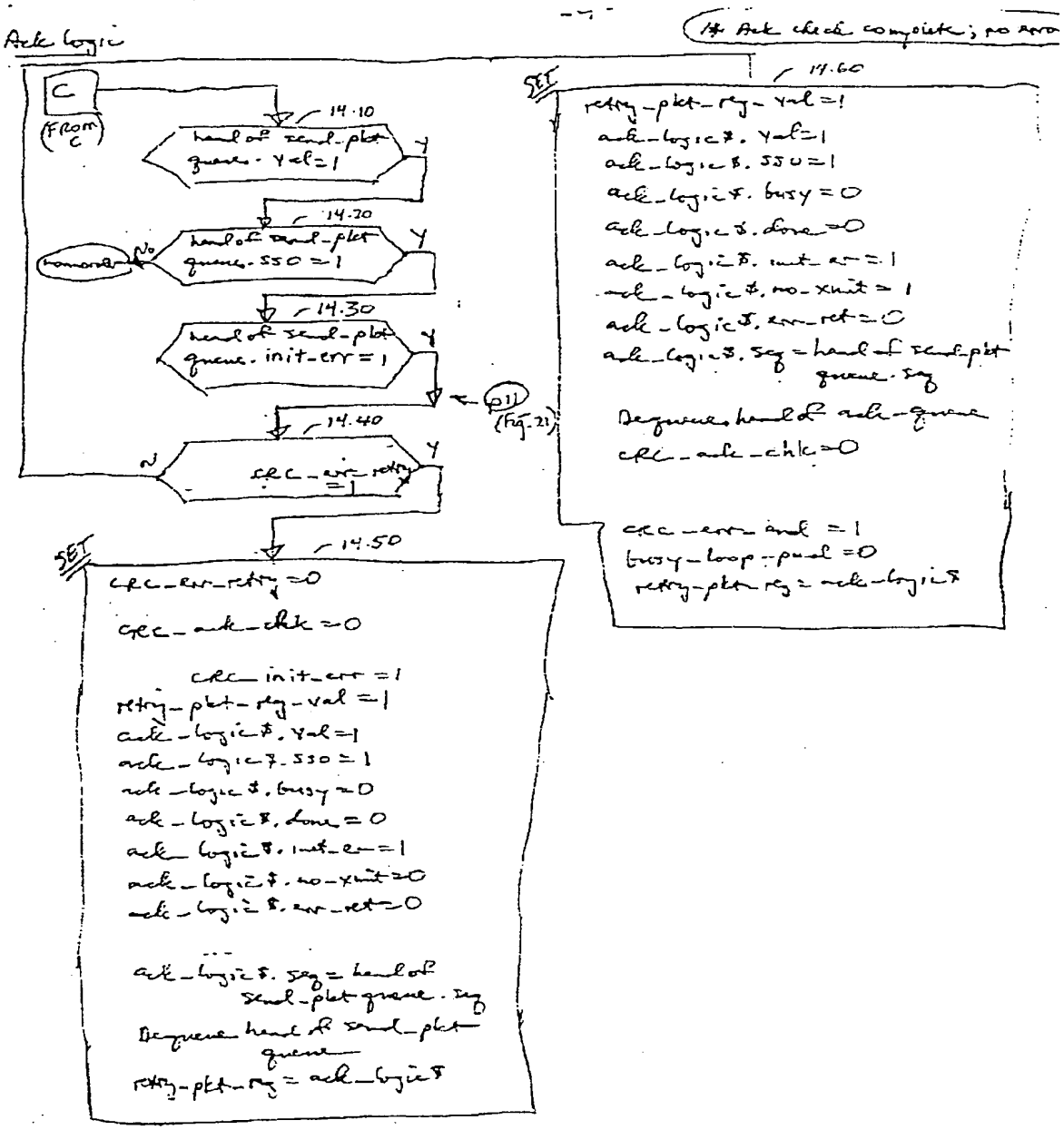
【Ｆｉｇ．１２】


Ack_logic
FROM A
A
12.10
head of send_pkt queue.val = 1
12.20
head of send_pkt queue.sso = 1
12.30
head of send_pkt queue.no_xmit = 1
12.40
head of ack queue .val = 1
12.50
head of ack_queue .sso = 1
12.60
head of send_pkt queue.seq = head of ack_queue.seq
12.70
idle_time out for head of send_pkt queue = 1
12.80
head of send_pkt queue.init_err = 1
12.90
retry_pkt_reg.val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 0
ack_logic$.no_xmit = 1
ack_logic$.err_ret = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_plt queue
leave head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
12.100
retry_pkt_reg.val = 1
ack_logic$.val = 1
ack_logic$.sso = 1
ack_logic$.busy = 0
ack_logic$.done = 0
ack_logic$.init_err = 1
ack_logic$.no_xmit = 0
ack_logic$.err_ret = 0
ack_logic$.seq = head of send_pkt queue.seq
Dequeue head of send_plt queue
leave head of ack_queue
log_err = 1
retry_pkt_reg = ack_logic$
12.110
head of send_pkt queue.init_err = 1
NO
YES
12.120
same as above left except replace "leave head of ack_queue" with "Dequeue head of ack_queue"
12.130
same as 12.100 except:
Ⓐ replace "leave head of ack_queue" with "Dequeue head of ack_queue"
Ⓑ change: ack_logic$.no_xmit = 1
Ⓒ add: clr.init_err = 0
Ⓓ add: clr_err = 1

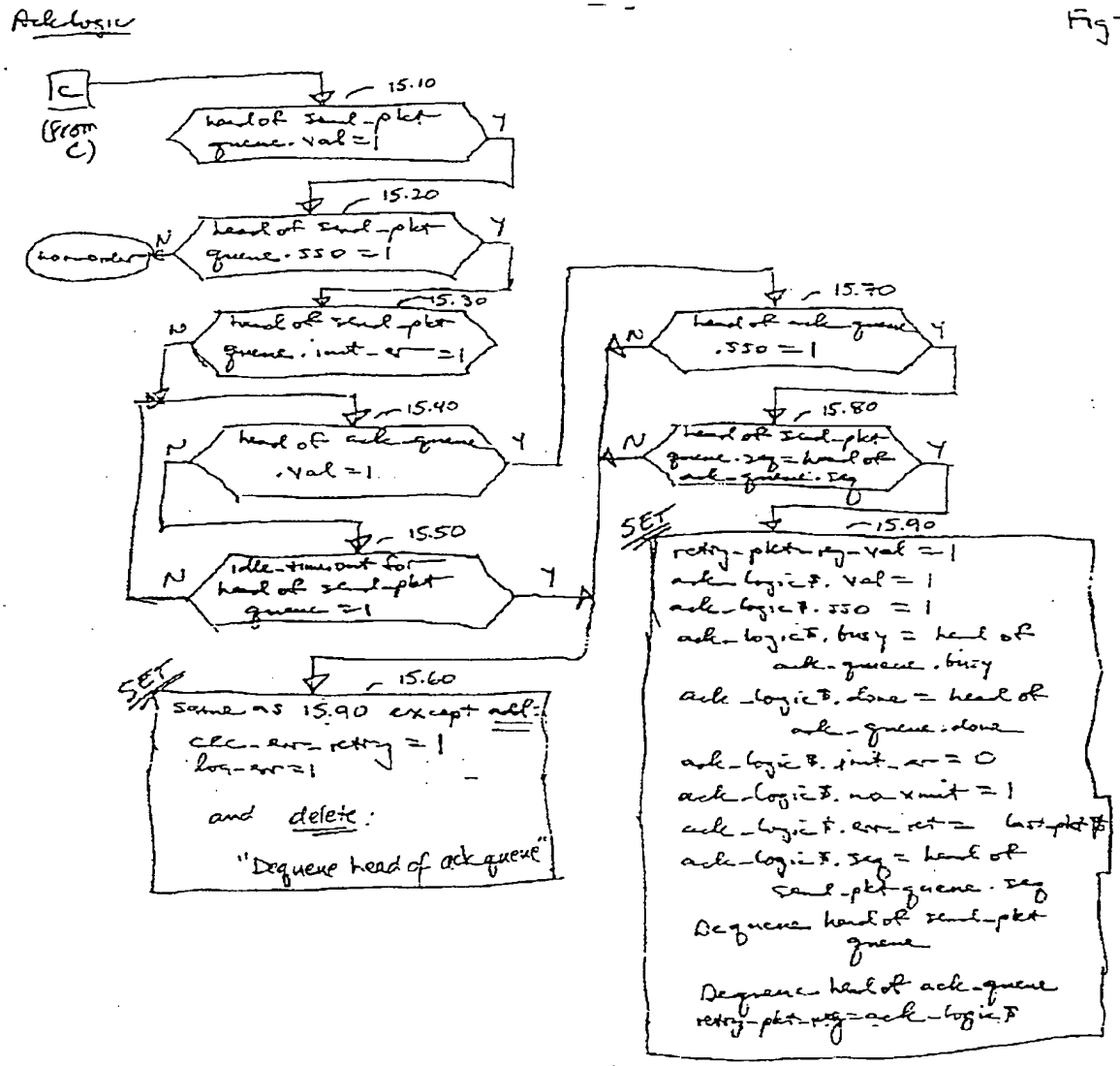
【Ｆｉｇ．１３】



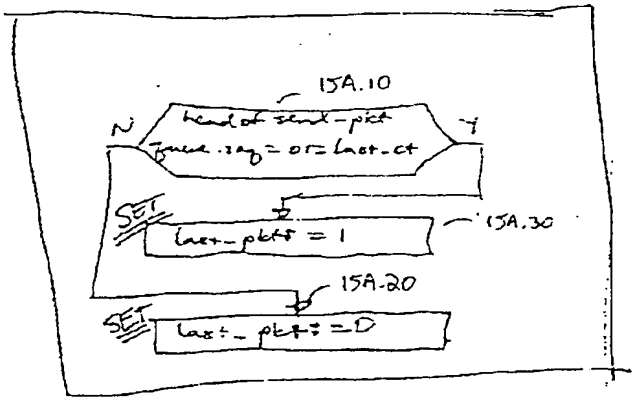
【Ｆｉｇ．１４】


Ack logic
C
(FROM C)
14.10
head of send-plct queue. val=1
14.20
head of send-plct queue. sso=1
14.30
head of send-plct queue. init-err=1
14.40
crc_err_retry =1
14.50
crc_err_retry=0
crc_ack_chk=0
crc_init_err=1
retry_plct_reg_val=1
ack_logic#. val=1
ack_logic#. sso=1
ack_logic#. busy=0
ack_logic#. done=0
ack_logic#. init_err=1
ack_logic#. no_xmit=0
ack_logic#. err_ret=0
ack_logic#. seq=head of send_plct queue. seq
Dequeue head of send_plct queue
retry_plct_reg=ack_logic#
SET
14.60
retry_plct_reg_val=1
ack_logic#. val=1
ack_logic#. sso=1
ack_logic#. busy=0
ack_logic#. done=0
ack_logic#. init_err=1
ack_logic#. no_xmit=1
ack_logic#. err_ret=0
ack_logic#. seq=head of send_plct queue. seq
Dequeue head of ack_queue
crc_ack_chk=0
crc_err_ack=1
busy_loop_proc=0
retry_plct_reg=ack_logic#
p11
(Fig. 21)

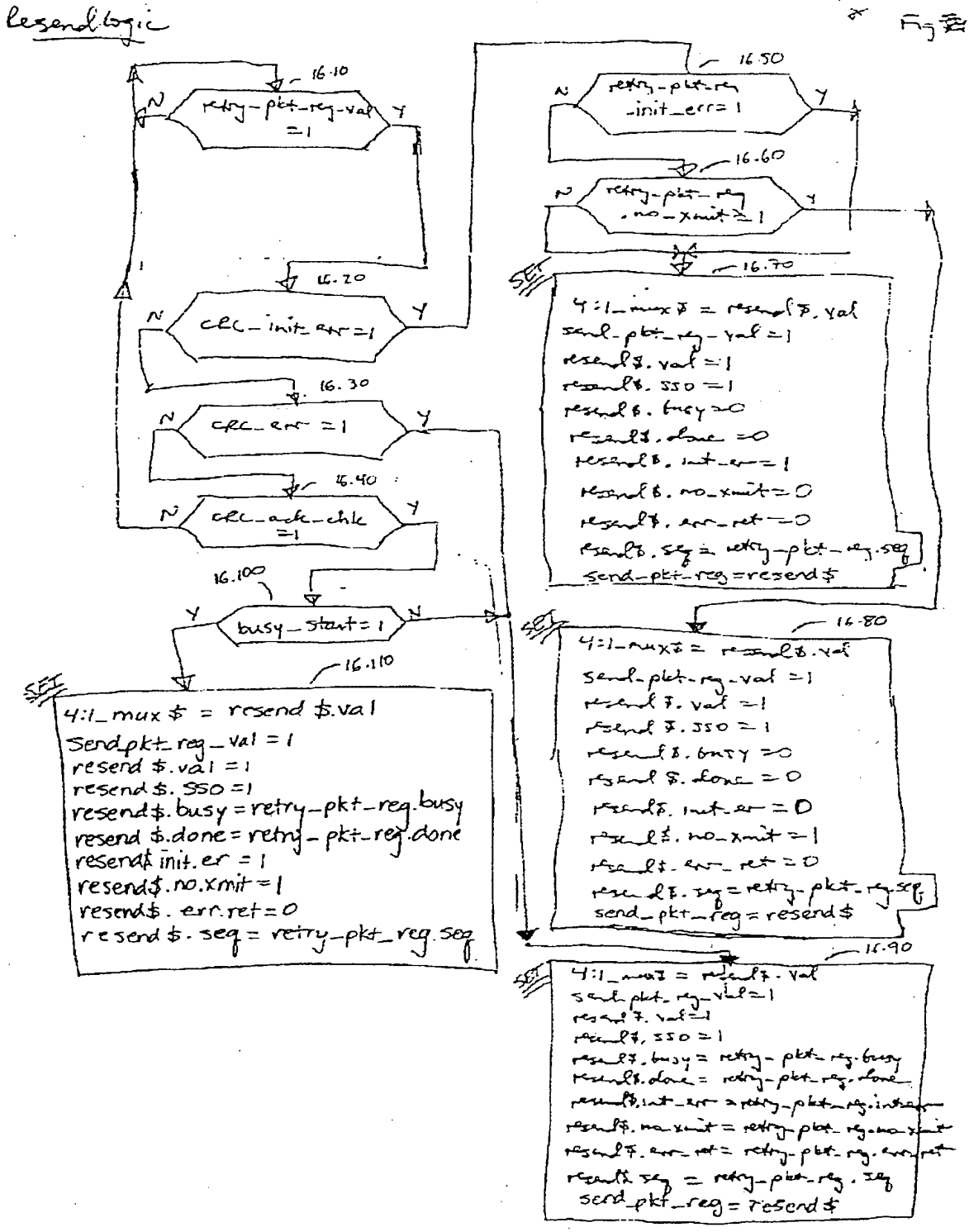
【Ｆｉｇ．１５】



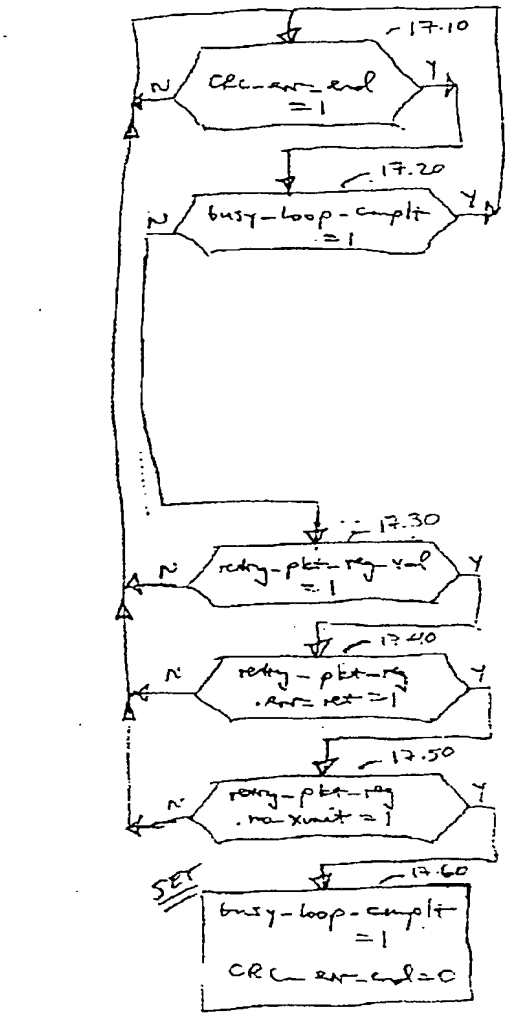
【Ｆｉｇ．１５Ａ】



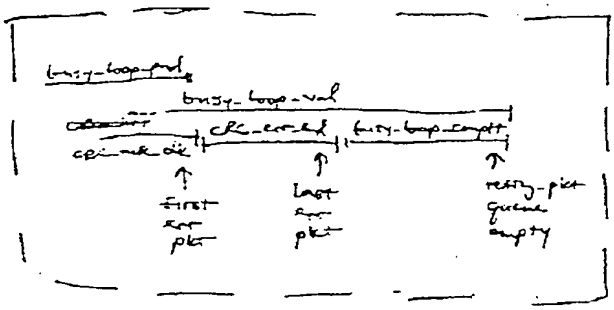
【Ｆｉｇ．１６】


resend logic
16.10
retry_pkt_req_val =1
16.20
crc_init_err=1
16.30
crc_err =1
16.40
crc_ack_chk =1
16.100
busy_start=1
16.110
4:1_mux$ = resend$.val
Send_pkt_req_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = retry_pkt_req.busy
resend$.done = retry_pkt_req.done
resend$.init.er = 1
resend$.no.xmit = 1
resend$.err.ret = 0
resend$.seq = retry_pkt_req.seq
16.50
retry_pkt_req_init_err=1
16.60
retry_pkt_req.no_xmit>=1
16.70
4:1_mux$ = resend$.val
send_pkt_req_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = 0
resend$.done = 0
resend$.init_er = 1
resend$.no_xmit = 0
resend$.err_ret = 0
resend$.seq = retry_pkt_req.seq
send_pkt_req = resend$
16.80
4:1_mux$ = resend$.val
send_pkt_req_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = 0
resend$.done = 0
resend$.init_er = 0
resend$.no_xmit = 1
resend$.err_ret = 0
resend$.seq = retry_pkt_req.seq
send_pkt_req = resend$
16.90
4:1_mux$ = resend$.val
send_pkt_req_val = 1
resend$.val = 1
resend$.sso = 1
resend$.busy = retry_pkt_req.busy
resend$.done = retry_pkt_req.done
resend$.init_err = retry_pkt_req.init_err
resend$.no_xmit = retry_pkt_req.no_xmit
resend$.err_ret = retry_pkt_req.err_ret
resend$.seq = retry_pkt_req.seq
send_pkt_req = resend$

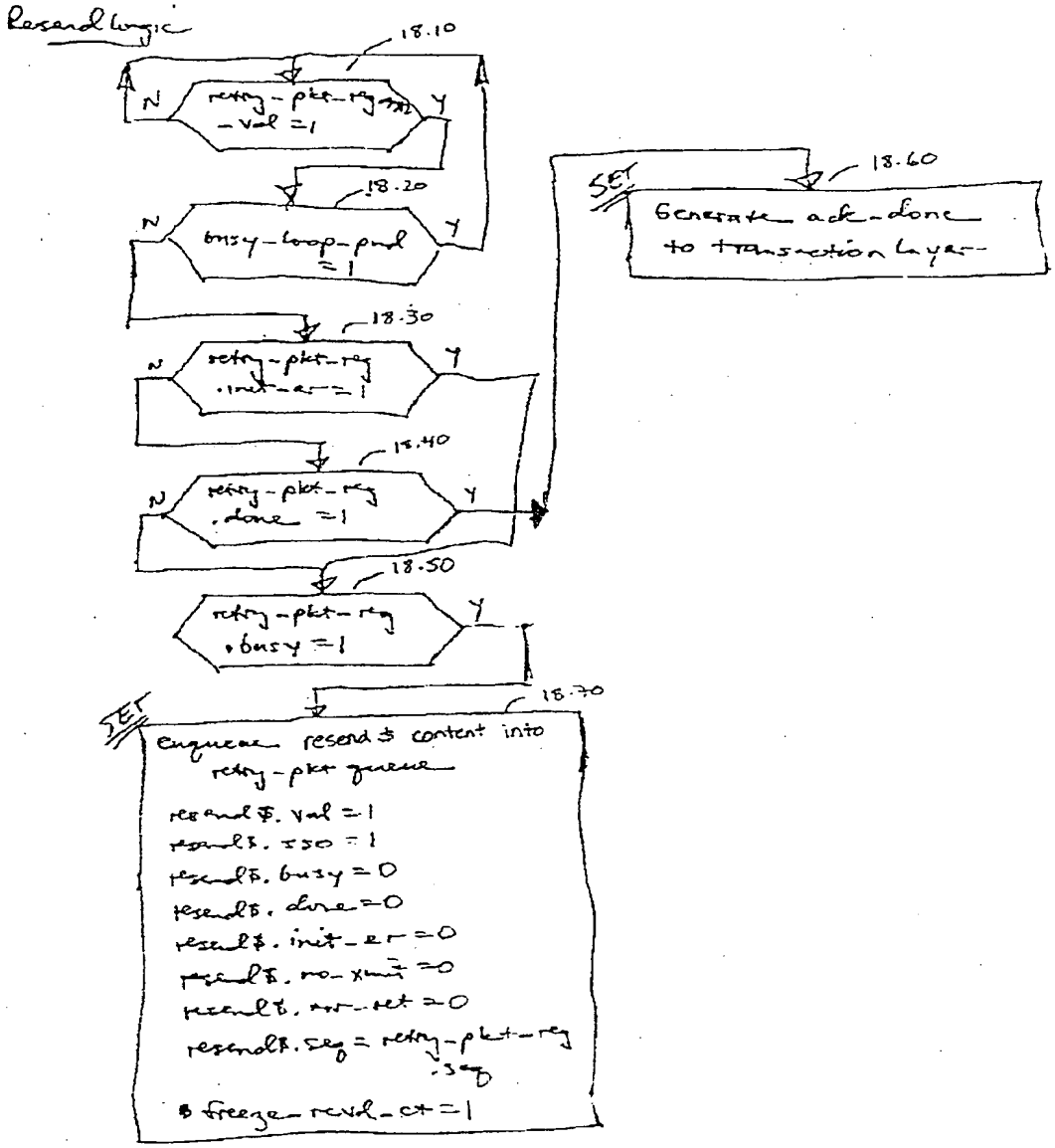
【Ｆｉｇ．１７】



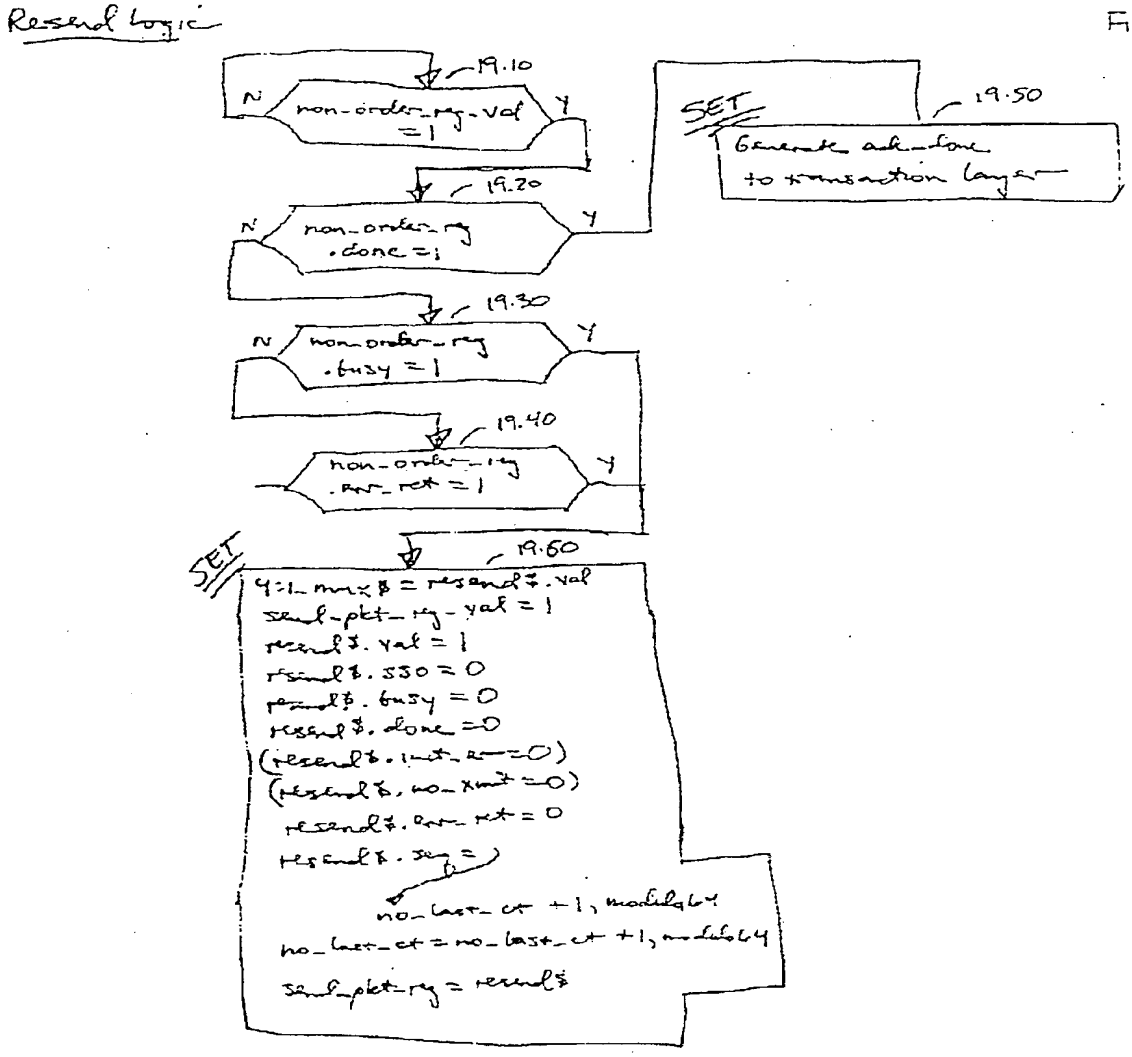
【Ｆｉｇ．１７Ａ】



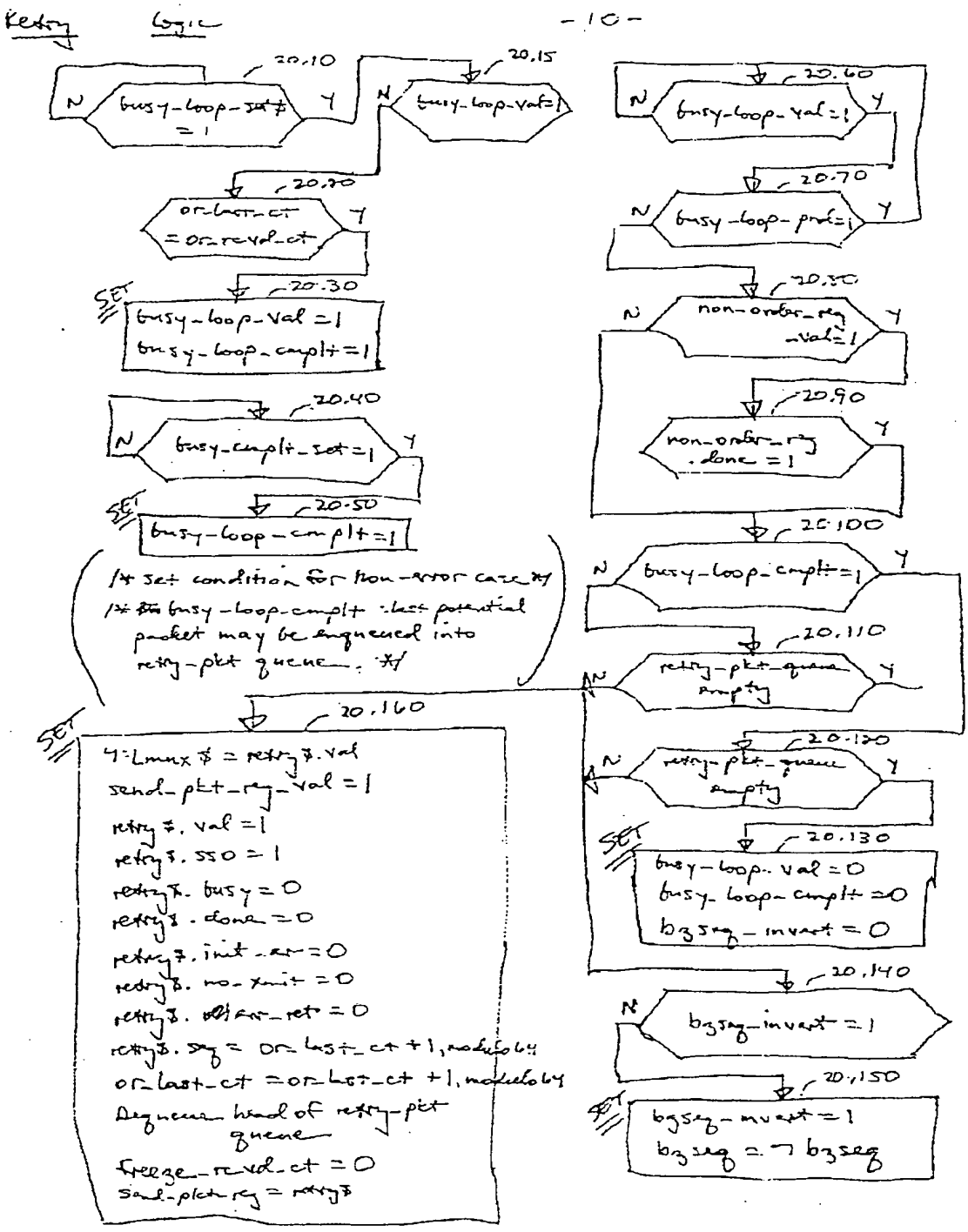
【Ｆｉｇ．１８】


Resend logic
18.10
retry-pkt-req .val =1
N
Y
18.20
busy-loop-prvl =1
18.30
retry-pkt-req .init-err=1
18.40
retry-pkt-req .done =1
18.50
retry-pkt-req .busy =1
18.60
SET
Generate ack-done to transaction layer
18.70
SET
enqueue resend$ content into retry-pkt queue
resend$.val =1
resend$.rso =1
resend$.busy =0
resend$.done=0
resend$.init-err=0
resend$.no-xmit=0
resend$.not-set=0
resend$.seq = retry-pkt-req .seq
$ freeze-rcvd-ct=1

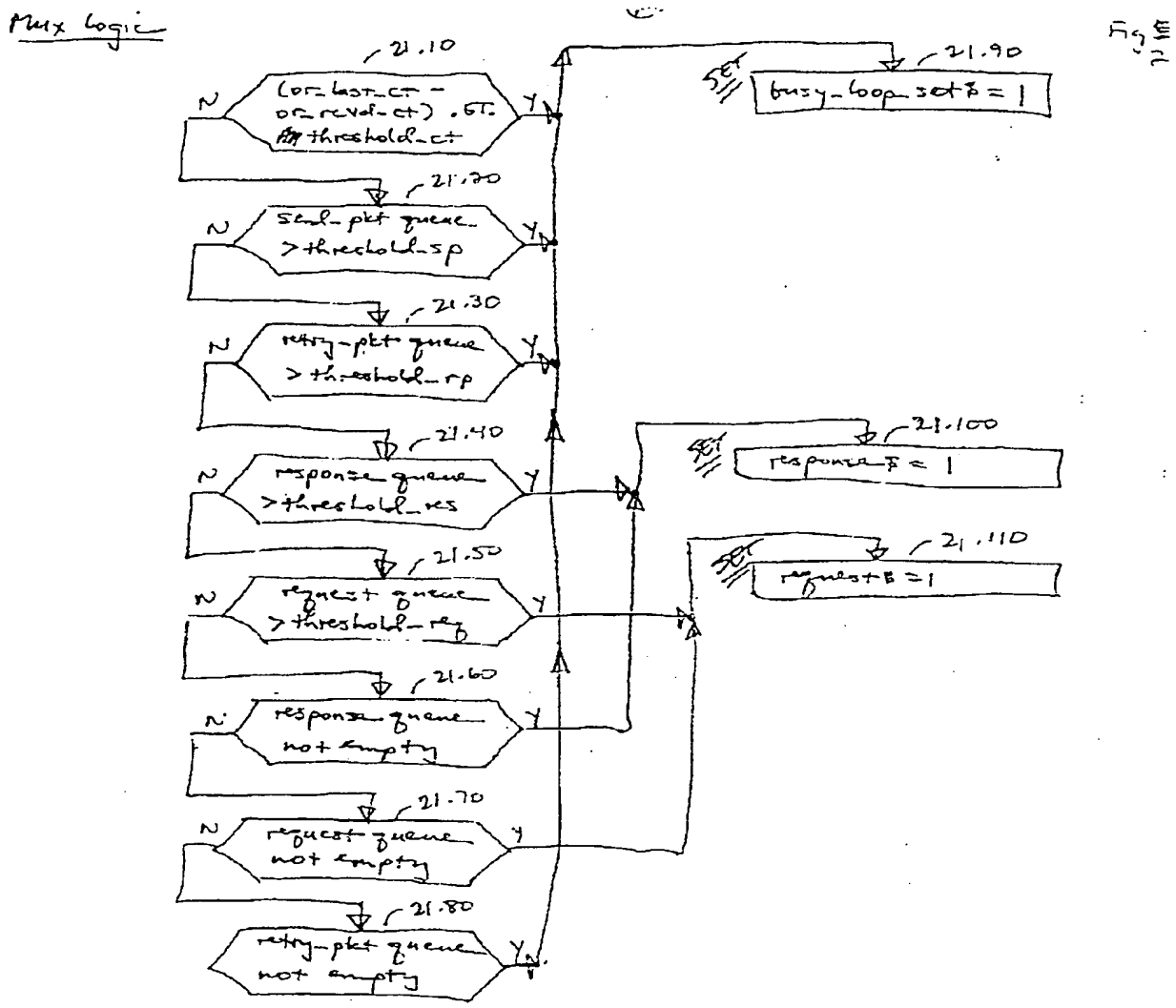
【Ｆｉｇ．１９】


Resend logic
19.10
N
non-order_rq.val = 1
Y
19.20
N
non-order_rq .done = 1
Y
SET
19.50
Generate ack_done to transaction layer
19.30
N
non-order_rq .busy = 1
Y
19.40
non-order_rq .err_ret = 1
Y
SET
19.60
q:1..max$ = resend$.val
send_pkt_rq.val = 1
resend$.val = 1
resend$.sso = 0
resend$.busy = 0
resend$.done = 0
(resend$.last_ack = 0)
(resend$.no_xmit = 0)
resend$.err_ret = 0
resend$.seq =
no_last_ack + 1, modulo64
no_last_ack = no_last_ack + 1, modulo64
send_pkt_rq = resend$

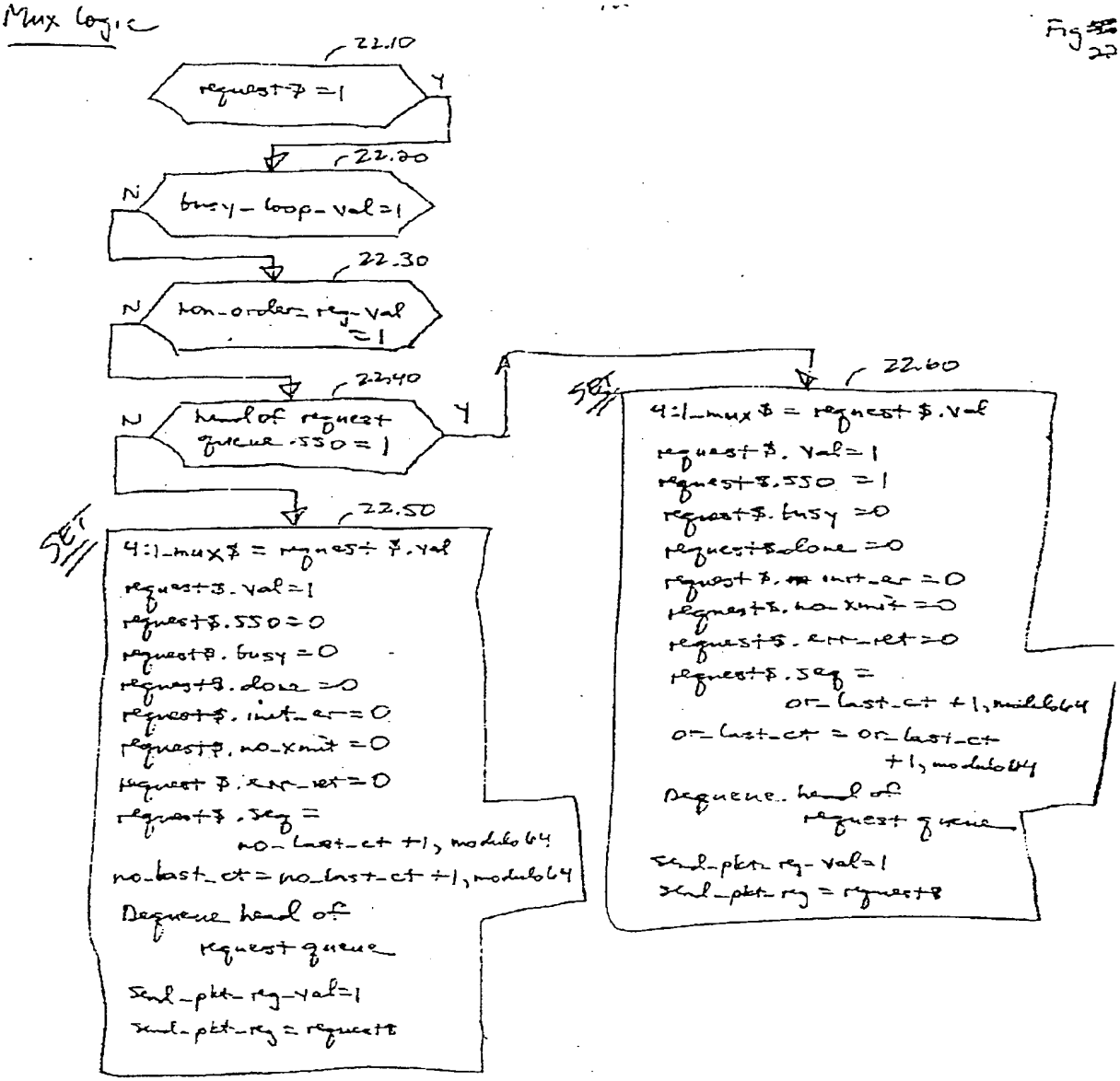
【Fig. 20】


Retry Logic
-10-
20.10
busy-loop-val=1
N
Y
20.15
busy-loop-val=1
N
20.20
or-last-ct = or-rcvd-ct
Y
SET
20.30
busy-loop-val=1
busy-loop-cmplt=1
20.40
busy-cmplt-set=1
N
Y
SET
20.50
busy-loop-cmplt=1
/* set condition for non-error case */
/* the busy-loop-cmplt : last potential packet may be enqueued into retry-pkt queue. */
SET
20.160
Y-Lmax # = retry#.val
send-pkt-rq-val=1
retry#.val=1
retry#.sso=1
retry#.busy=0
retry#.done=0
retry#.init-err=0
retry#.no-xmit=0
retry#.all-err-ret=0
retry#.seq = or-last-ct+1, modulo 64
or-last-ct = or-last-ct+1, modulo 64
Dequeue head of retry-pkt queue
freeze-rcvd-ct=0
send-pkt-rq = retry#
20.60
busy-loop-val=1
N
Y
20.70
busy-loop-prot=1
N
Y
20.80
non-order-rq-val=1
N
Y
20.90
non-order-rq.done=1
Y
20.100
busy-loop-cmplt=1
N
Y
20.110
retry-pkt-queue empty
N
Y
20.120
retry-pkt-queue empty
N
Y
SET
20.130
busy-loop-val=0
busy-loop-cmplt=0
bgseq-invert=0
20.140
bgseq-invert=1
N
20.150
bgseq-invert=1
bgseq = ¬ bgseq

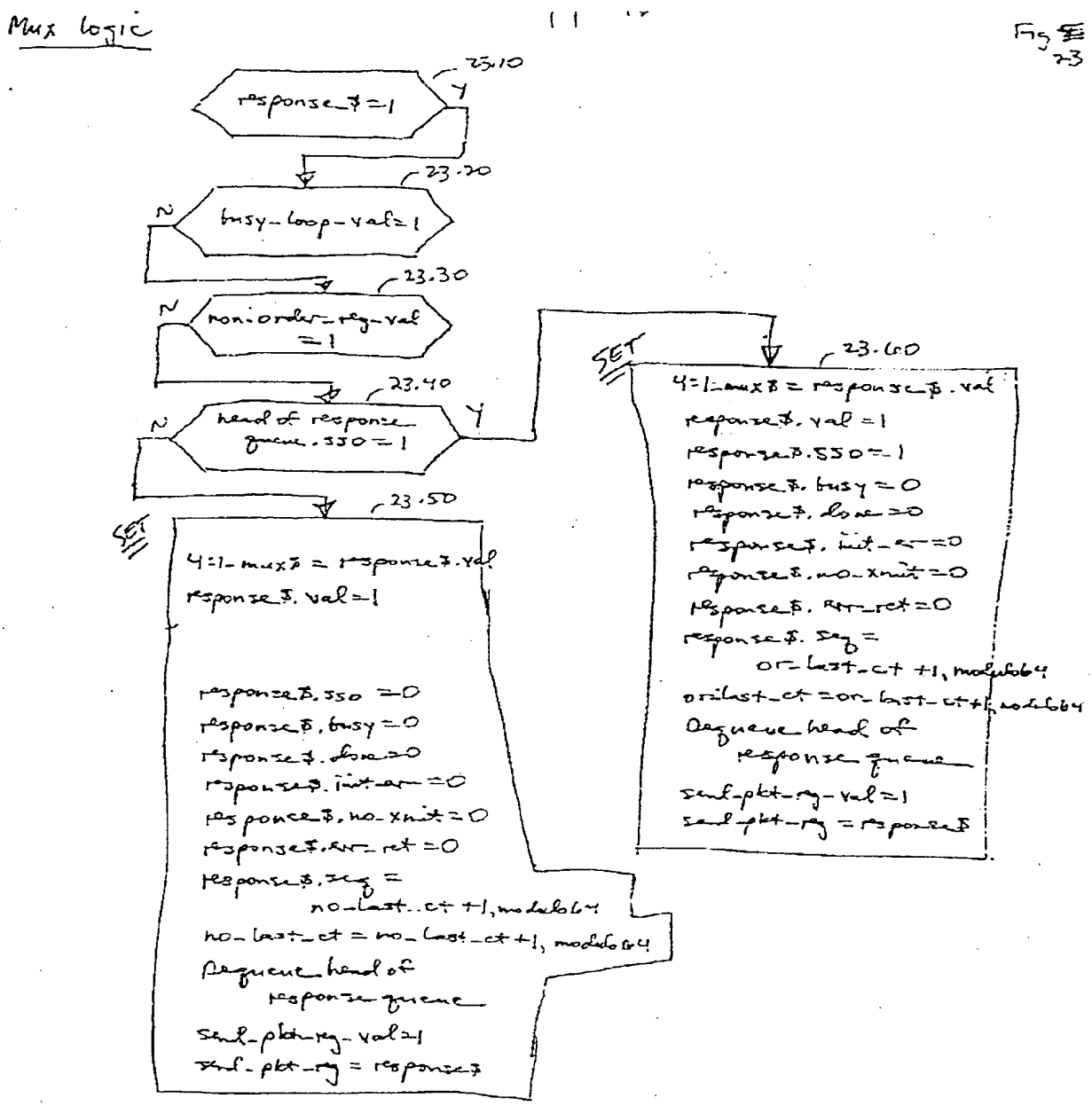
【Ｆｉｇ．２１】


Mux Logic
21.10
(or_last_ct - or_rcvd_ct) .GT. threshold_ct
N
Y
21.90
busy_loop_set = 1
21.20
send_pkt queue > threshold_sp
21.30
retry_pkt queue > threshold_rp
21.40
response queue > threshold_res
21.100
response = 1
21.50
request queue > threshold_req
21.110
request = 1
21.60
response queue not empty
21.70
request queue not empty
21.80
retry_pkt queue not empty

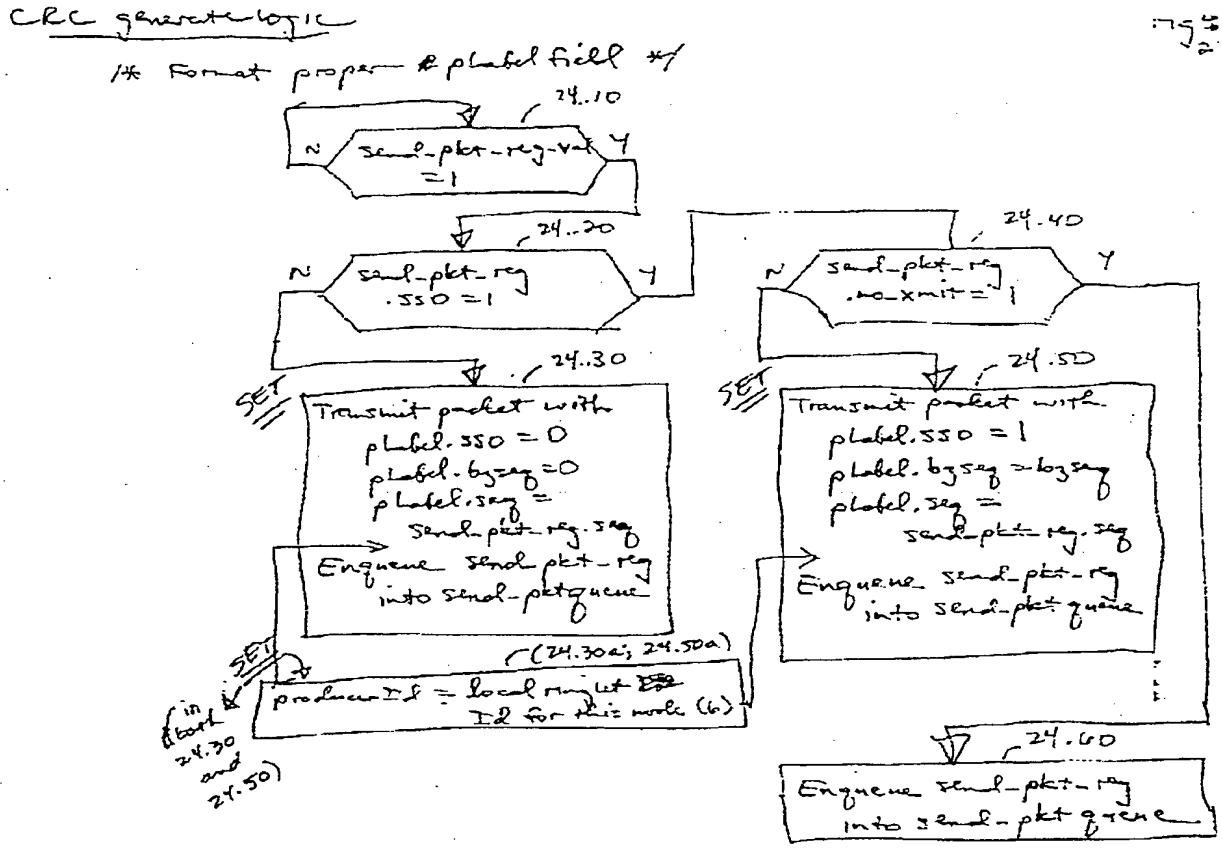
【Ｆｉｇ．２２】


Mux logic
Fig 22
22.10
request# = 1
Y
22.20
N
busy_loop_val = 1
22.30
N
non_order_req_val = 1
22.40
N
head of request queue.SSO = 1
Y
A
22.50
4:1_mux# = request#.val
request#.val = 1
request#.SSO = 0
request#.busy = 0
request#.done = 0
request#.init_er = 0
request#.no_xmit = 0
request#.err_ret = 0
request#.seq = no_last_ct + 1, modulo64
no_last_ct = no_last_ct + 1, modulo64
Dequeue head of request queue
send_pkt_req_val = 1
send_pkt_req = request#
22.60
4:1_mux# = request#.val
request#.val = 1
request#.SSO = 1
request#.busy = 0
request#.done = 0
request#.init_er = 0
request#.no_xmit = 0
request#.err_ret = 0
request#.seq = or_last_ct + 1, modulo64
or_last_ct = or_last_ct + 1, modulo64
Dequeue head of request queue
send_pkt_req_val = 1
send_pkt_req = request#

【Ｆｉｇ．２３】


Mux logic
Fig 23
23.10
response_$=1
Y
23.20
N
busy_loop_val=1
23.30
N
non-order_reg_val =1
23.40
N
head of response queue.sso=1
Y
SET
23.50
4=1-mux$ = response$.val
response_$.val=1
response$.sso =0
response$.busy=0
response$.done=0
response$.init_err=0
response$.no_xmit=0
response$.rtr_ret=0
response$.seq = no_last_ct+1, modulo64
no_last_ct = no_last_ct+1, modulo64
Dequeue head of response queue
send_pkt_reg_val=1
send_pkt_reg = response$
SET
23.60
4=1-mux$ = response_$.val
response$.val=1
response$.sso=1
response$.busy=0
response$.done=0
response$.init_err=0
response$.no_xmit=0
response$.rtr_ret=0
response$.seq = or_last_ct+1, modulo64
or_last_ct = or_last_ct+1, modulo64
Dequeue head of response queue
send_pkt_reg_val=1
send_pkt_reg = response$

【Ｆｉｇ．２４】

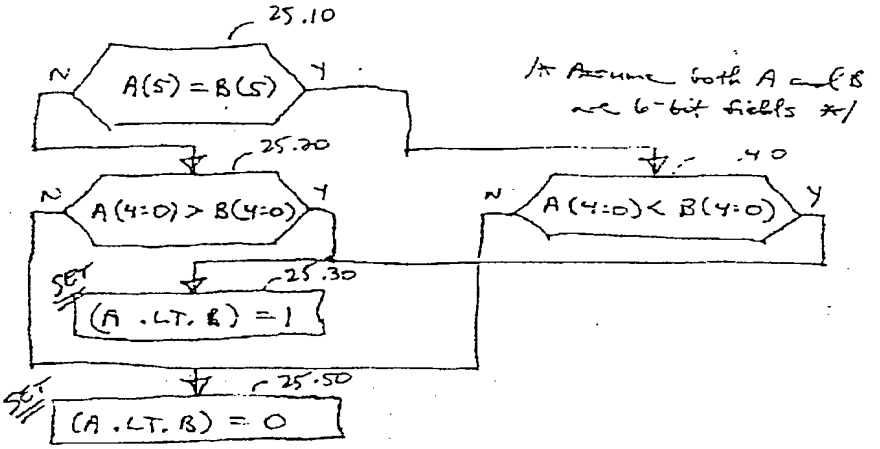

CRC generate logic
/* Format proper plabel field */
24..10
N
send_pkt_req_val
=1
Y
24..20
N
send_pkt_req
.sso =1
Y
24.40
N
send_pkt_req
.no_xmit = 1
Y
24..30
SET
Transmit packet with
plabel.sso = 0
plabel.bgseq=0
plabel.seq =
send_pkt_req.seq
Enqueue send_pkt_req
into send_pktqueue
24.50
SET
Transmit packet with
plabel.sso = 1
plabel.bgseq = bgseq
plabel.seq =
send_pkt_req.seq
Enqueue send_pkt_req
into send_pkt queue
(24.30a; 24.50a)
producer Id = local ring let Id for this work (6)
SET
(in front 24.30 and 24.50)
24.60
Enqueue send_pkt_req
into send_pkt queue

【Ｆｉｇ．２５】

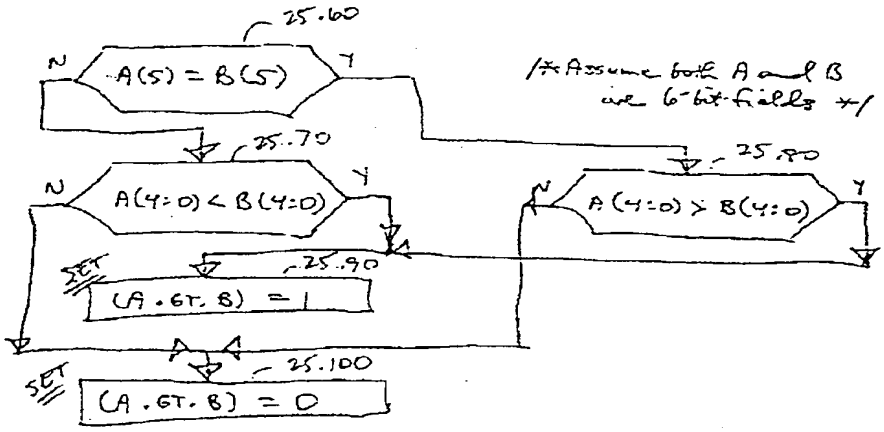
Fig 25

Comparison logic

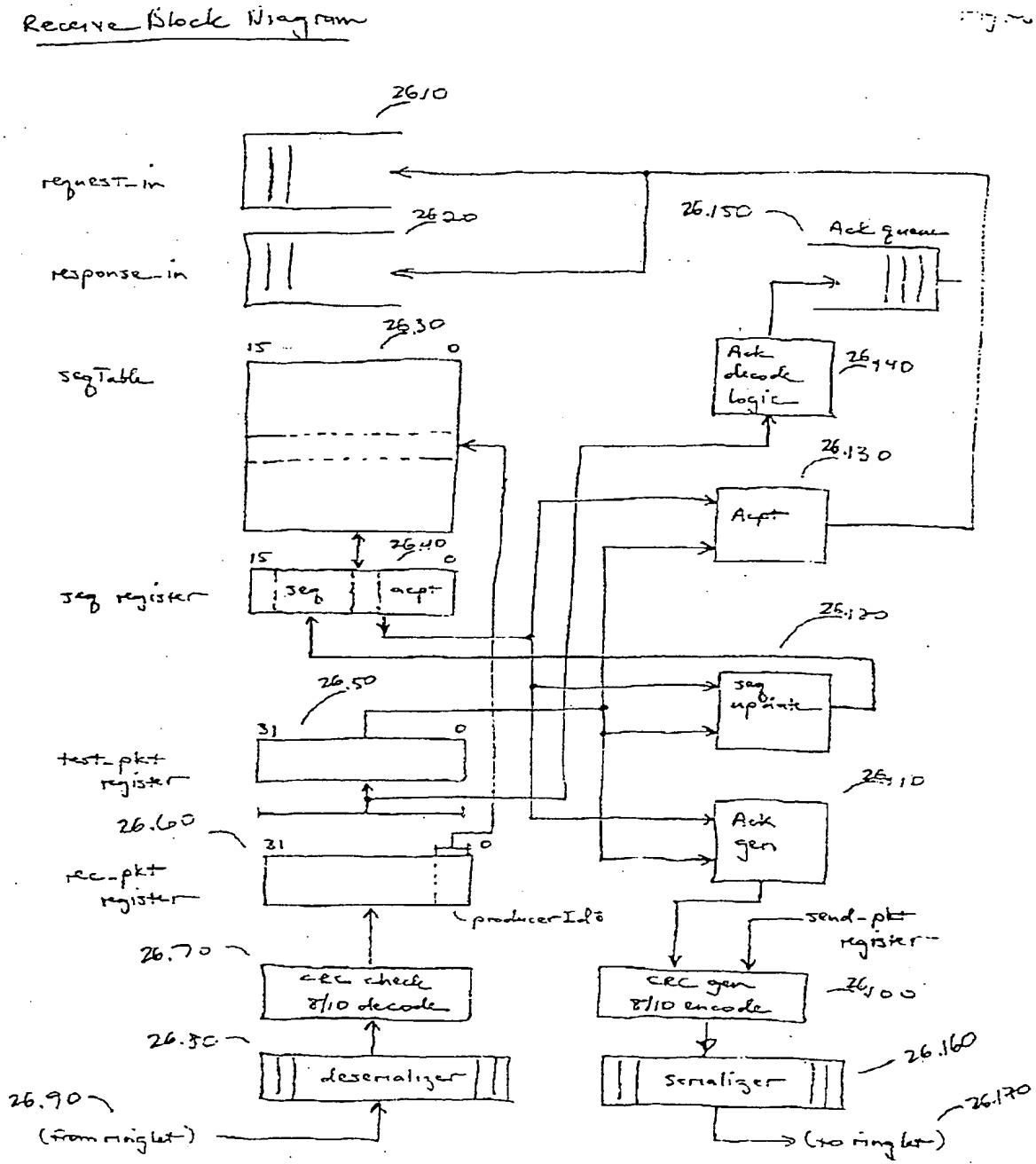
A .LT. B



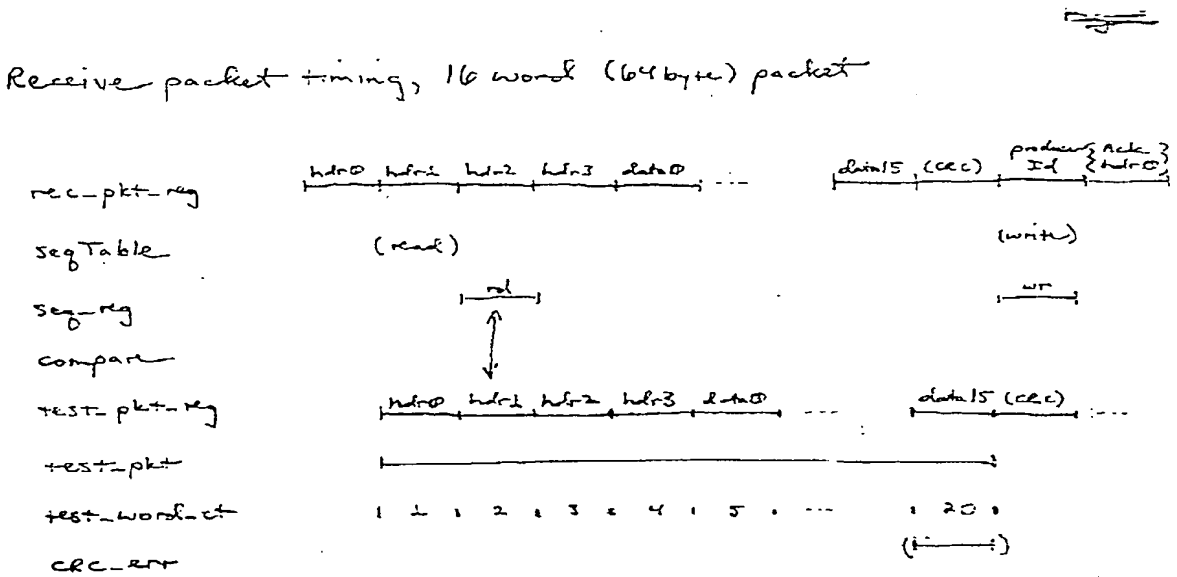
A .GT. B



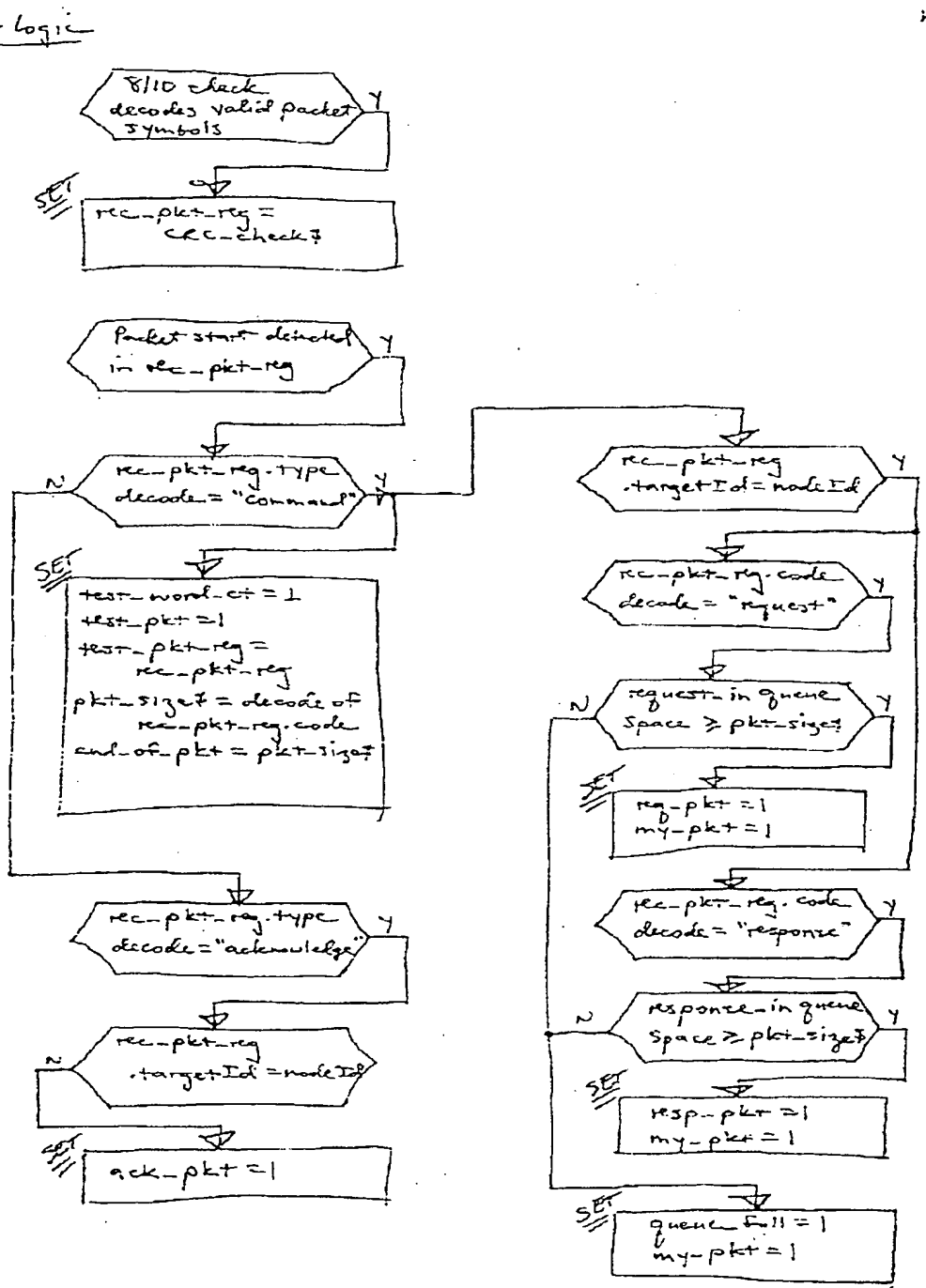
【Ｆｉｇ．２６】


Receive Block Diagram
26.10
request_in
26.20
response_in
26.150
Ack queue
26.30
15
0
seqTable
Ack decode logic
26.140
26.130
Acpt
15
0
seq register
seq
acpt
26.120
26.50
31
0
seq update
test_pkt register
26.110
26.60
31
0
Ack gen
rec_pkt register
producerId
send_pkt register
26.70
CRC check 8/10 decode
CRC gen 8/10 encode
26.100
26.80
deserializer
serializer
26.160
26.90
(from ringlet)
26.170
(to ringlet)

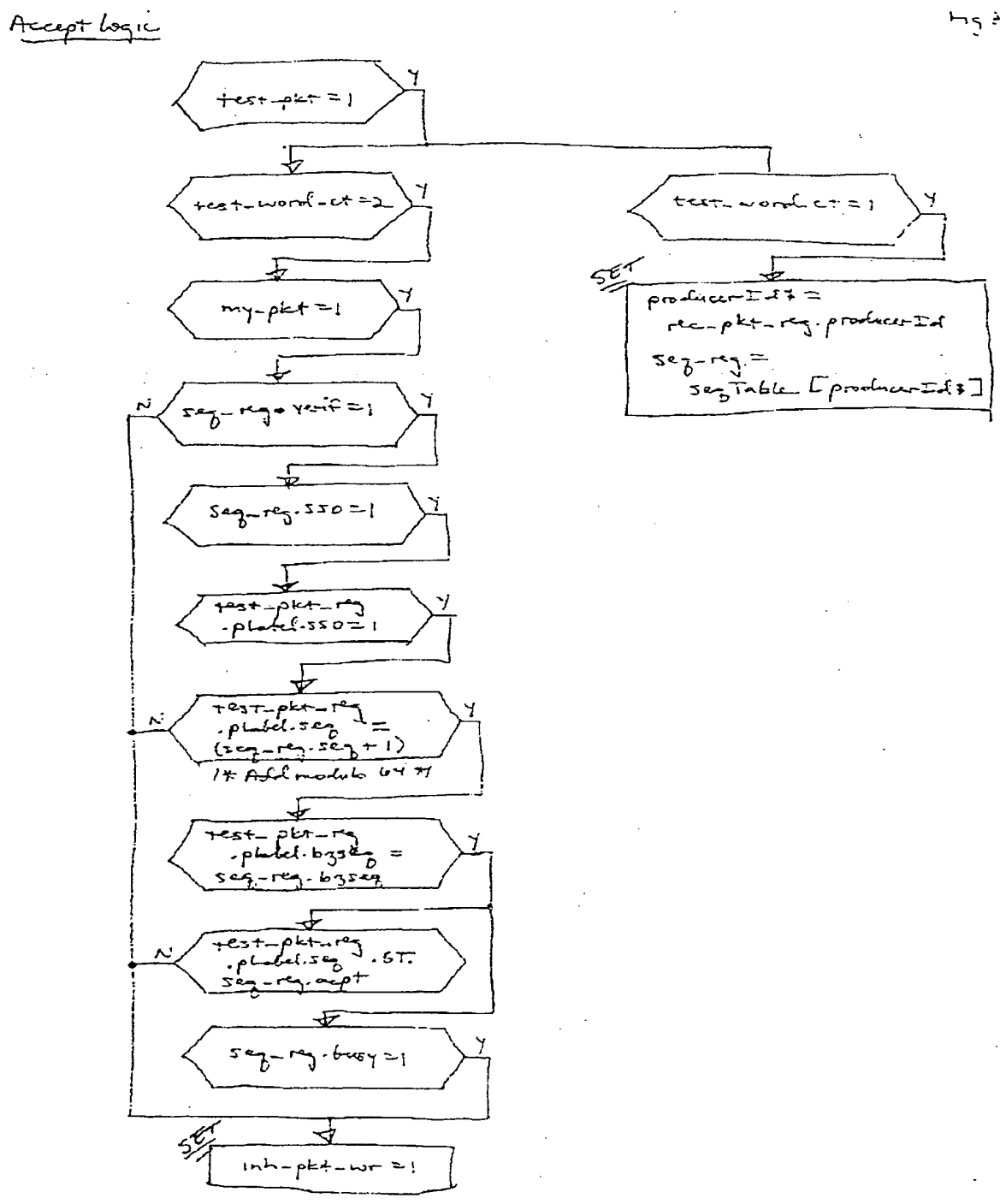
【Ｆｉｇ．２７Ａ】


Receive packet timing, 16 word (64 byte) packet
rec_pkt_reg
hdr0 hdr1 hdr2 hdr3 data0
data15 (crc)
Ack_hdr0
seqTable
(read)
(write)
seq_reg
rd
wr
compare
test_pkt_reg
test_pkt
test_word_ct
1 2 3 4 5 20
crc_err

【Ｆｉｇ．２８】


Accept logic
8/10 check decodes valid packet symbols
Y
SET
rec_pkt_reg = crc_check?
Packet start detected in rec_pkt_reg
Y
rec_pkt_reg.type decode = "command"
N
Y
SET
test_word_ct = 1
test_pkt = 1
test_pkt_reg = rec_pkt_reg
pkt_size? = decode of rec_pkt_reg.code
end_of_pkt = pkt_size?
rec_pkt_reg.type decode = "acknowledge"
Y
rec_pkt_reg .targetId = nodeId?
N
SET
ack_pkt = 1
rec_pkt_reg .targetId = nodeId?
Y
rec_pkt_reg.code decode = "request"
Y
request_in queue Space ≥ pkt_size?
N
Y
SET
req_pkt = 1
my_pkt = 1
rec_pkt_reg.code decode = "response"
Y
response_in queue Space ≥ pkt_size?
N
Y
SET
resp_pkt = 1
my_pkt = 1
SET
queue_full = 1
my_pkt = 1

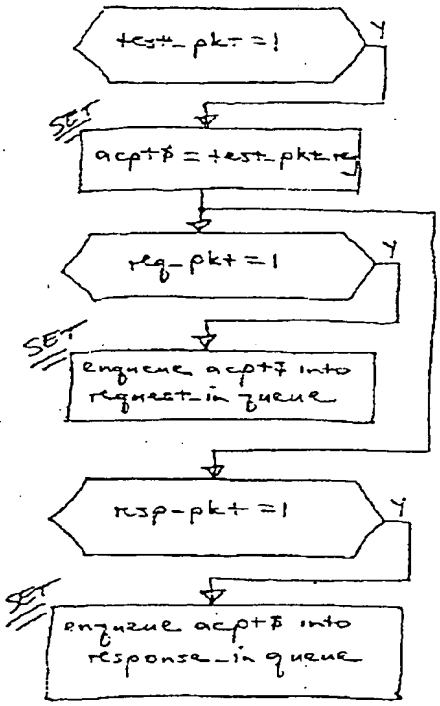
【Ｆｉｇ．２９】

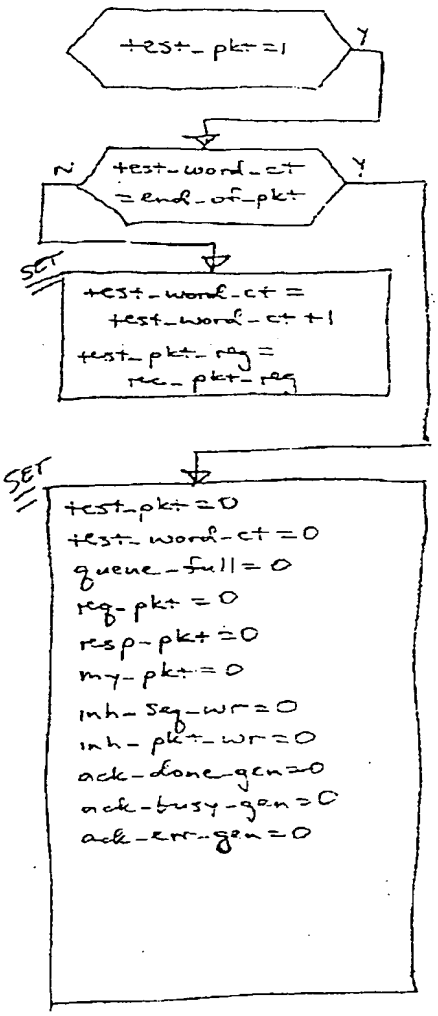

Accept logic
test_pkt = 1
test_word_ct = 2
my_pkt = 1
seq_reg.vsif = 1
seq_reg.SSO = 1
test_pkt_reg .plabel.SSO = 1
test_pkt_reg .plabel.seq0 = (seq_reg.seq0 + 1)
/* Add modulo 64 */
test_pkt_reg .plabel.bgseq0 = seq_reg.bgseq
test_pkt_reg .plabel.seq0 .GT. seq_reg.acpt
seq_reg.busy = 1
SET
inh_pkt_wr = 1
test_word_ct = 1
SET
producerId# = rec_pkt_reg.producerId
seq_reg = seqTable [producerId#]
Y
N

【Ｆｉｇ．３０】

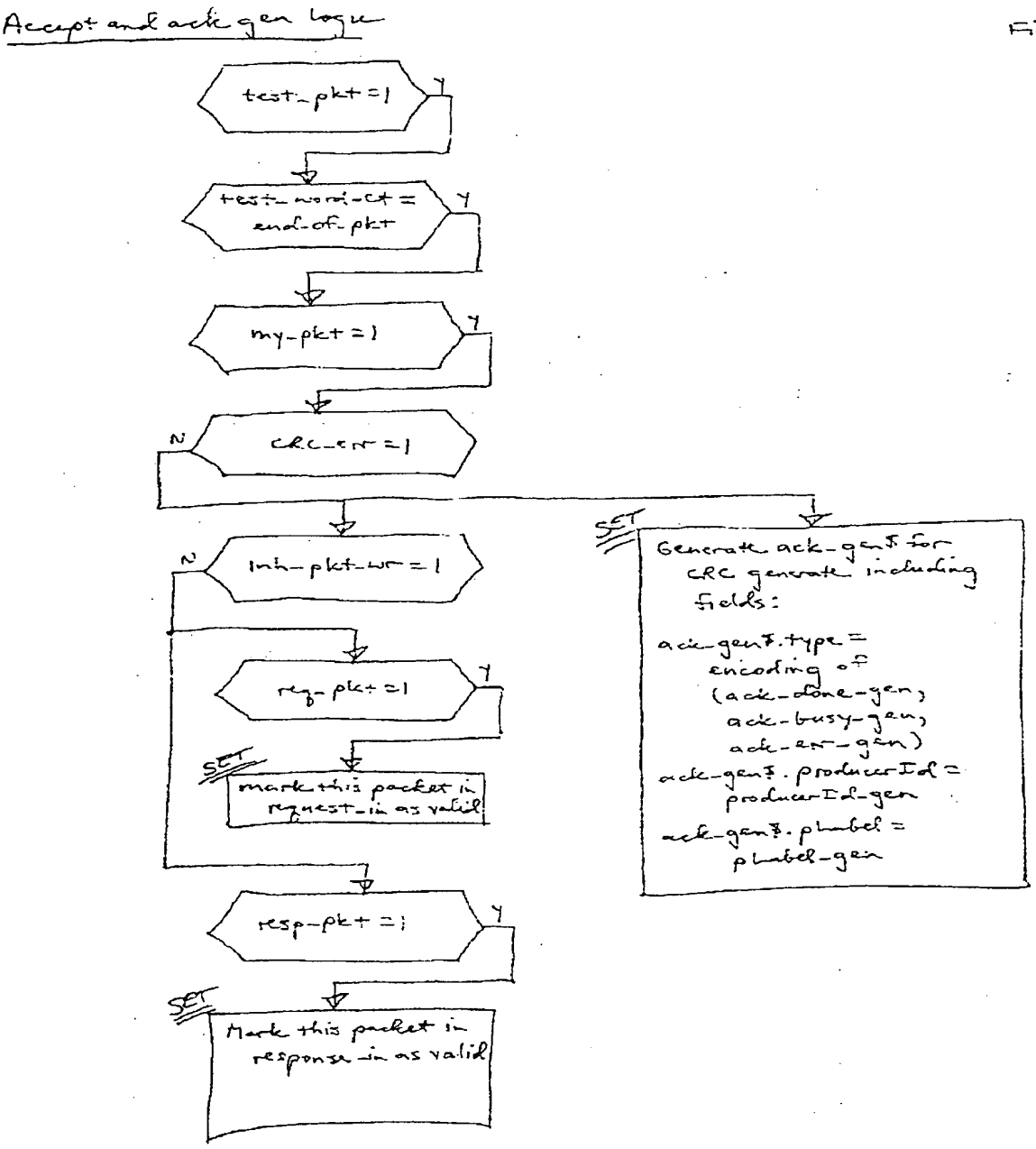
Accept and ack_gen logic

Fig 30

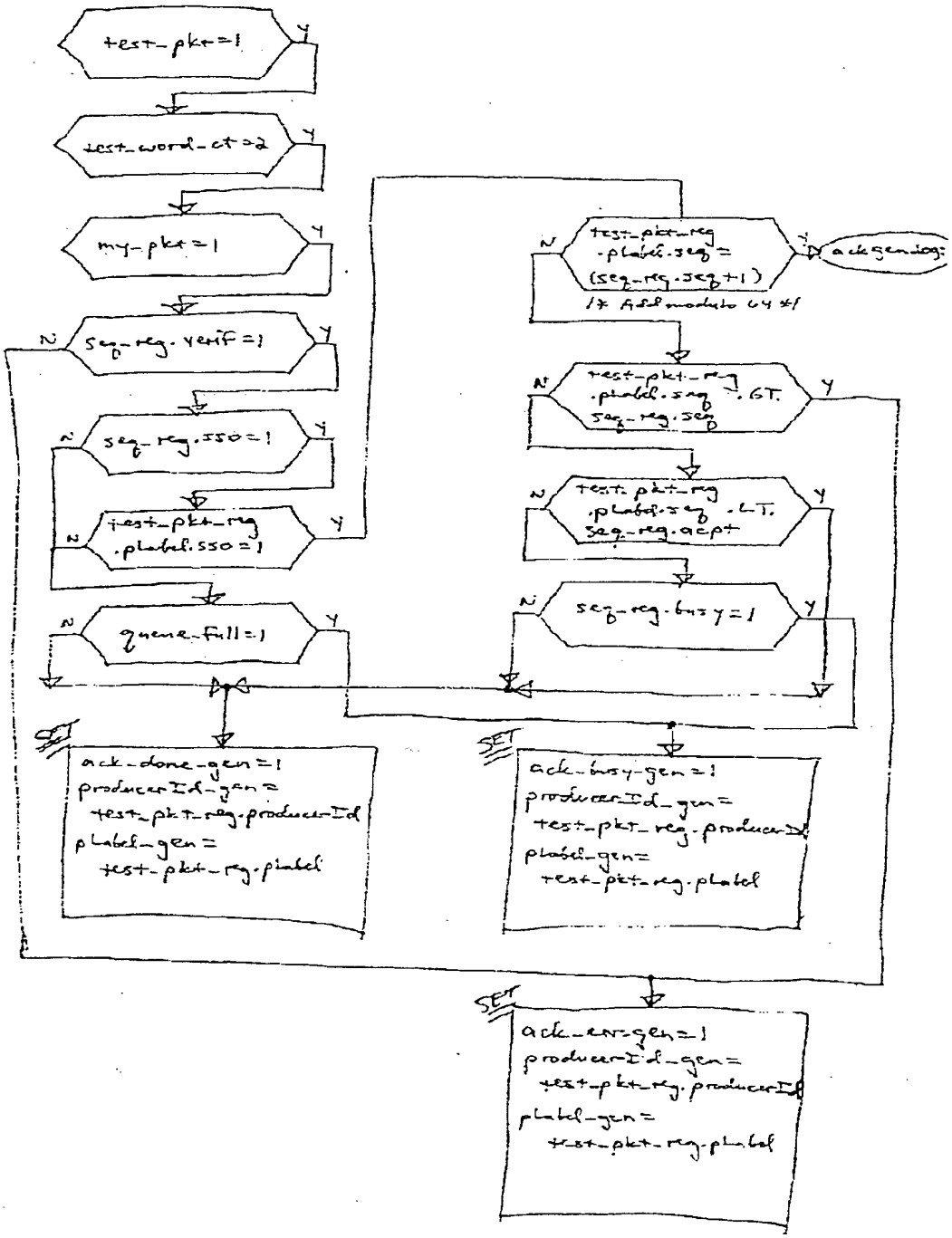




【Ｆｉｇ．３１】


Accept and ack gen logic
Fig 31
test_pkt = 1
Y
test_word_ct = end_of_pkt
Y
my_pkt = 1
Y
N
CRC_err = 1
N
Inh_pkt_wr = 1
req_pkt = 1
Y
SET
mark this packet in request_in as valid
resp_pkt = 1
Y
SET
Mark this packet in response_in as valid
SET
Generate ack_gen# for CRC generate including fields:
ack_gen#.type = encoding of (ack_done_gen, ack_busy_gen, ack_err_gen)
ack_gen#.producerId = producerId_gen
ack_gen#.plabel = plabel_gen

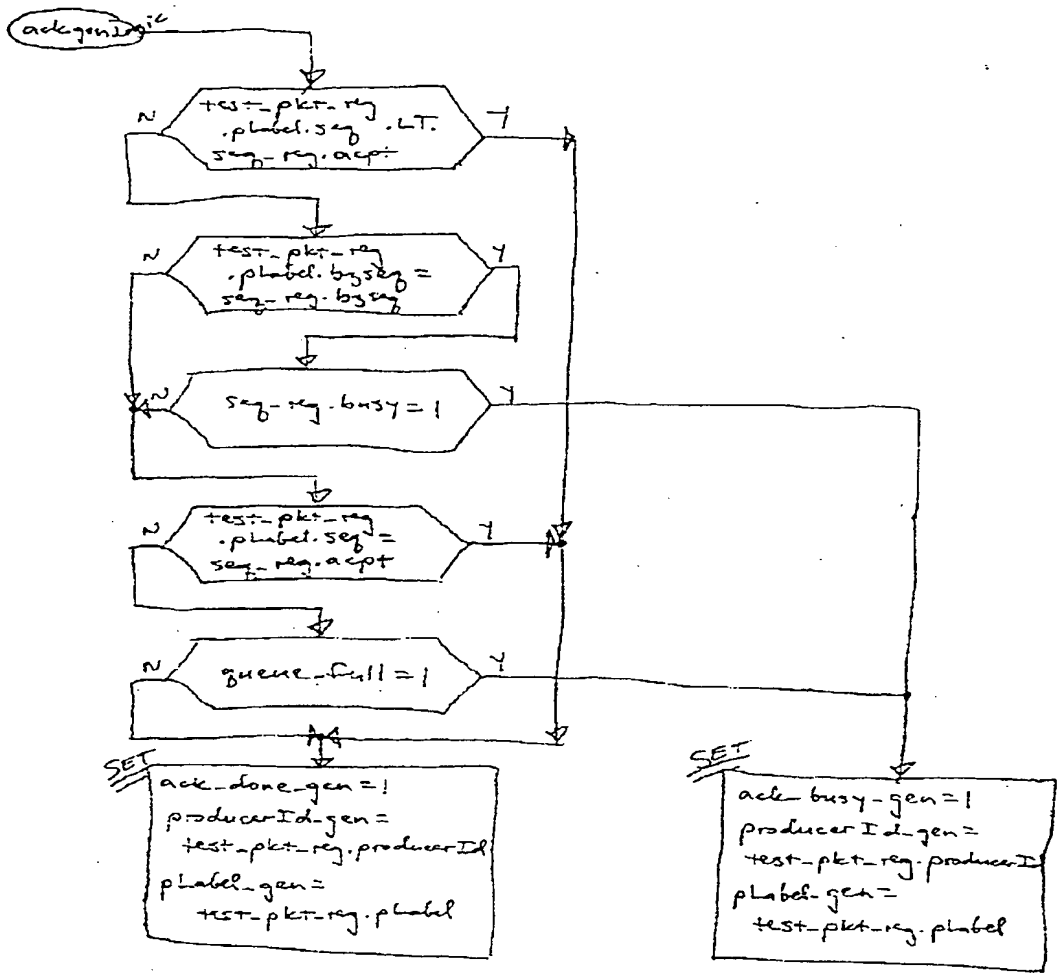
【Ｆｉｇ．３２】


Ack gen logic
Fig 3
test_pkt=1
test_word_ct=2
my_pkt=1
seq_reg.verif=1
seq_reg.sso=1
test_pkt_reg.plabel.sso=1
queue_full=1
test_pkt_reg.plabel.seq = (seq_reg.seq+1)
/* Add modulo 64 */
ack gen logic
test_pkt_reg.plabel.seq > GT. seq_reg.seq
test_pkt_reg.plabel.seq < LT. seq_reg.acpt
seq_reg.busy=1
SET
ack_done_gen=1
producerId_gen=
test_pkt_reg.producerId
plabel_gen=
test_pkt_reg.plabel
SET
ack_busy_gen=1
producerId_gen=
test_pkt_reg.producerId
plabel_gen=
test_pkt_reg.plabel
SET
ack_err_gen=1
producerId_gen=
test_pkt_reg.producerId
plabel_gen=
test_pkt_reg.plabel

【Ｆｉｇ．３３】

Ack gen logic

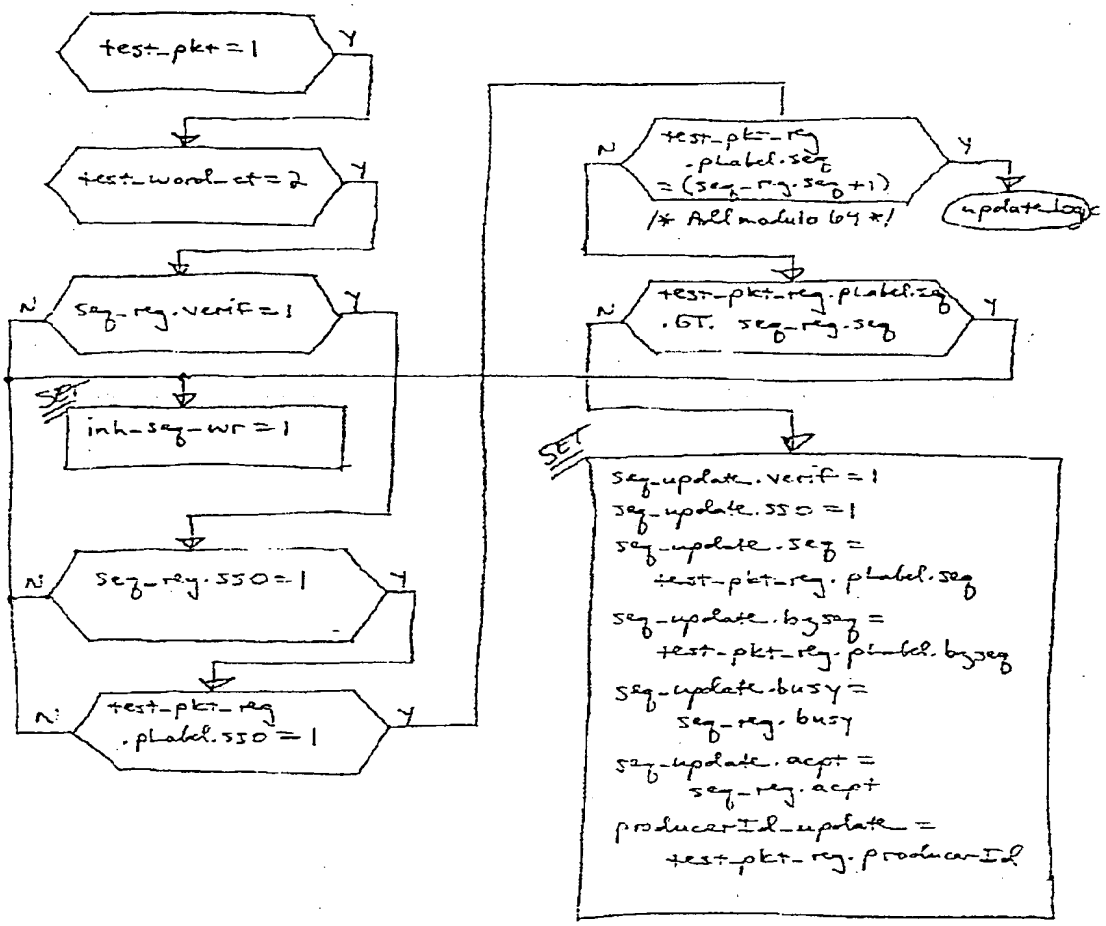
Fig 33



【Ｆｉｇ．３４】

Seq update

Fig 34



【Fig. 35】

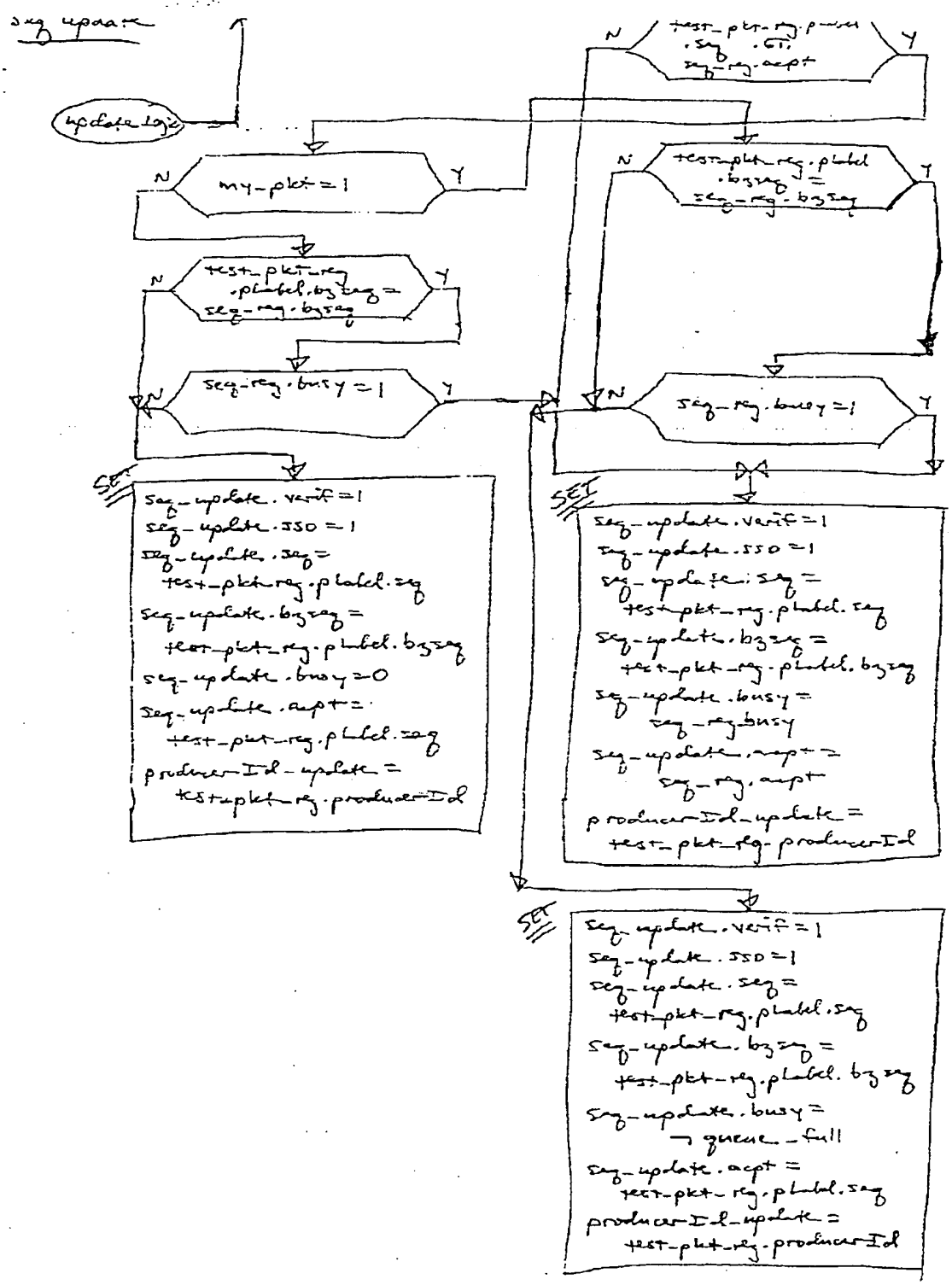
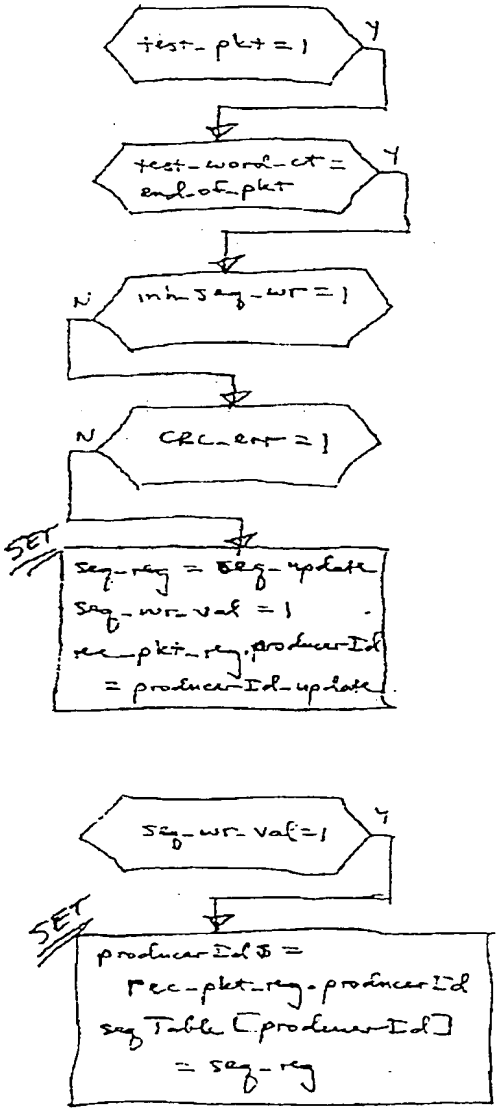

update logic
my-pkt=1
test-pkt-reg.plabel.bgseq=seq-reg.bgseq
seq-reg.busy=1
test-pkt-reg.plabel.seq
seq-reg.acpt
test-pkt-reg.plabel.bgseq=seq-reg.bgseq
seq-reg.busy=1
SET
seq-update.verif=1
seq-update.sso=1
seq-update.seq=test-pkt-reg.plabel.seq
seq-update.bgseq=test-pkt-reg.plabel.bgseq
seq-update.busy=0
seq-update.acpt=test-pkt-reg.plabel.seq
producerId-update=test-pkt-reg.producerId
SET
seq-update.verif=1
seq-update.sso=1
seq-update.seq=test-pkt-reg.plabel.seq
seq-update.bgseq=test-pkt-reg.plabel.bgseq
seq-update.busy=seq-reg.busy
seq-update.acpt=seq-reg.acpt
producerId-update=test-pkt-reg.producerId
SET
seq-update.verif=1
seq-update.sso=1
seq-update.seq=test-pkt-reg.plabel.seq
seq-update.bgseq=test-pkt-reg.plabel.bgseq
seq-update.busy=¬queue_full
seq-update.acpt=test-pkt-reg.plabel.seq
producerId-update=test-pkt-reg.producerId

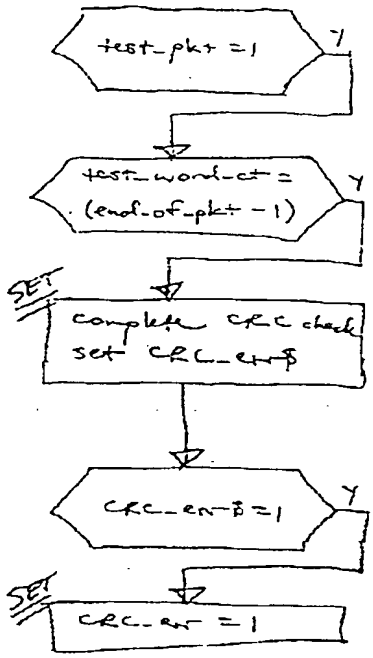
Fig 35

【Ｆｉｇ．３６】

Seq_D update

Fig 36

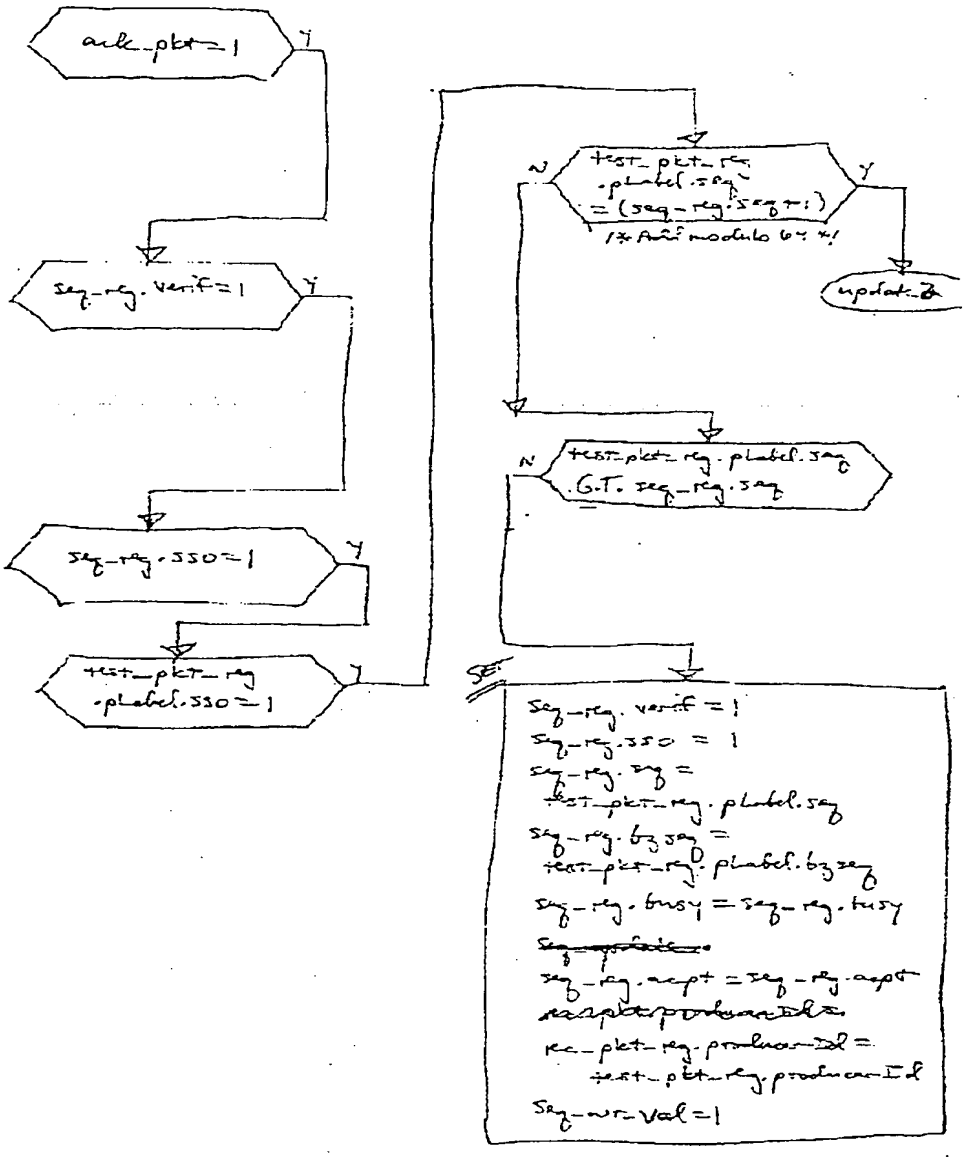




【Fig.37】

Seq update

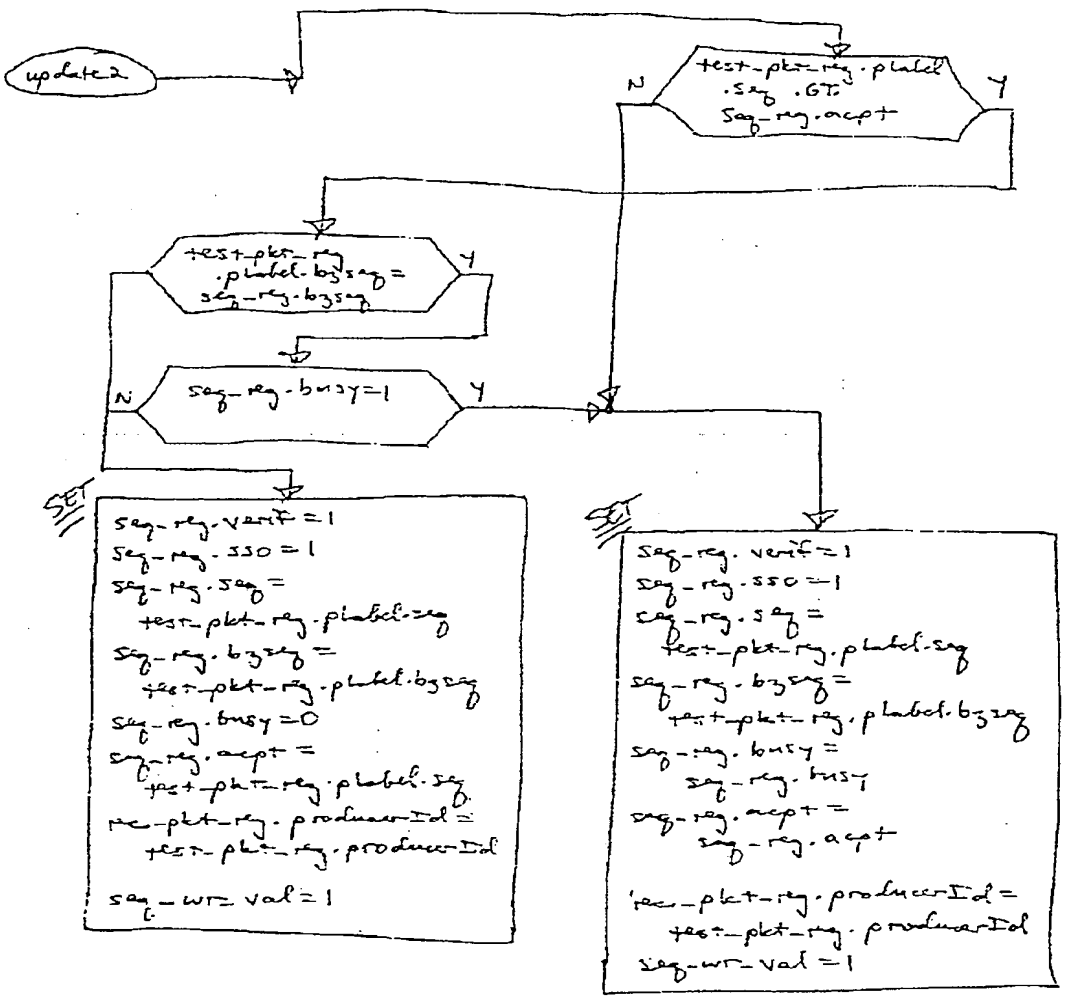
Fig 37



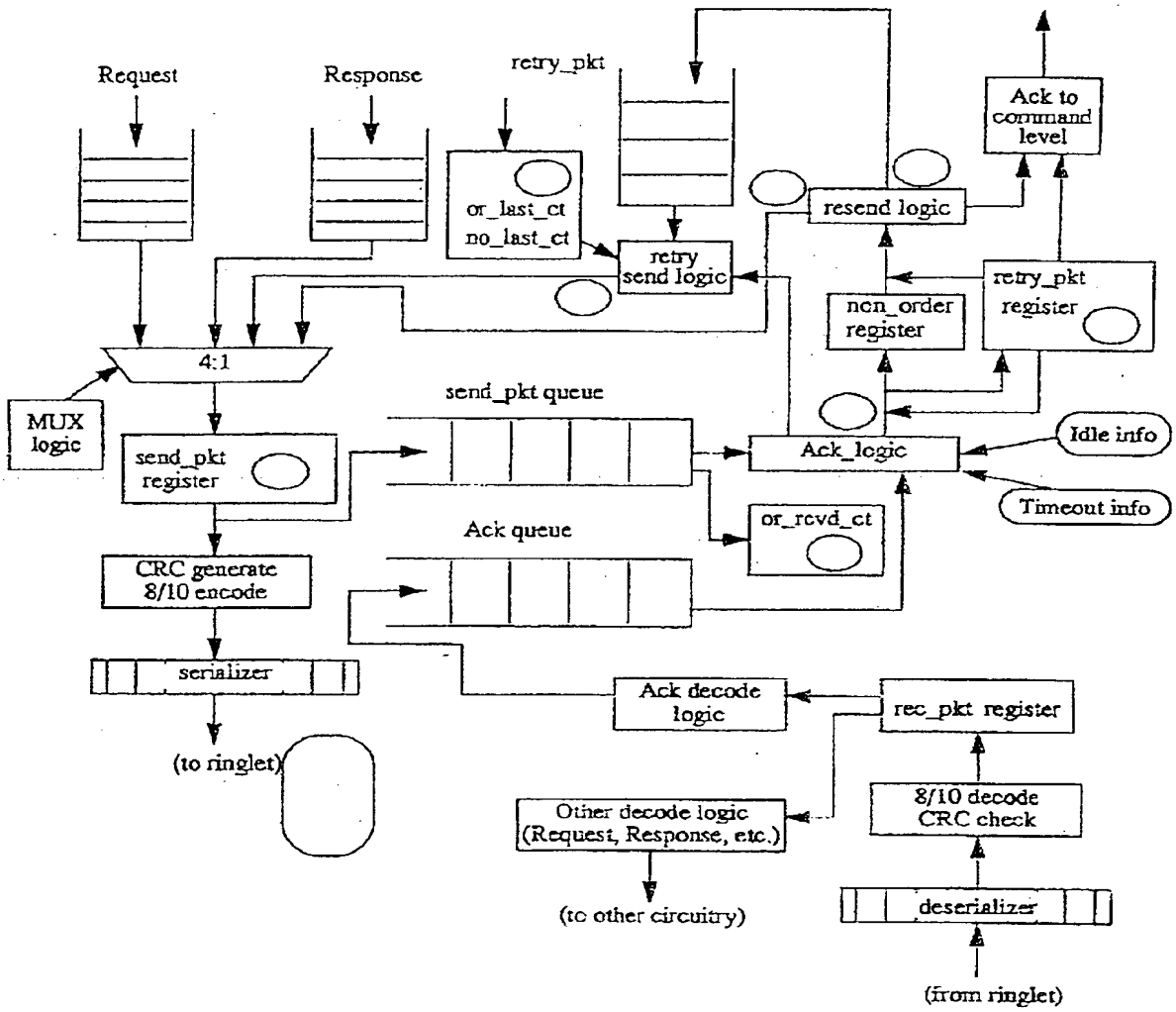
【Ｆｉｇ．３８】

Seq update

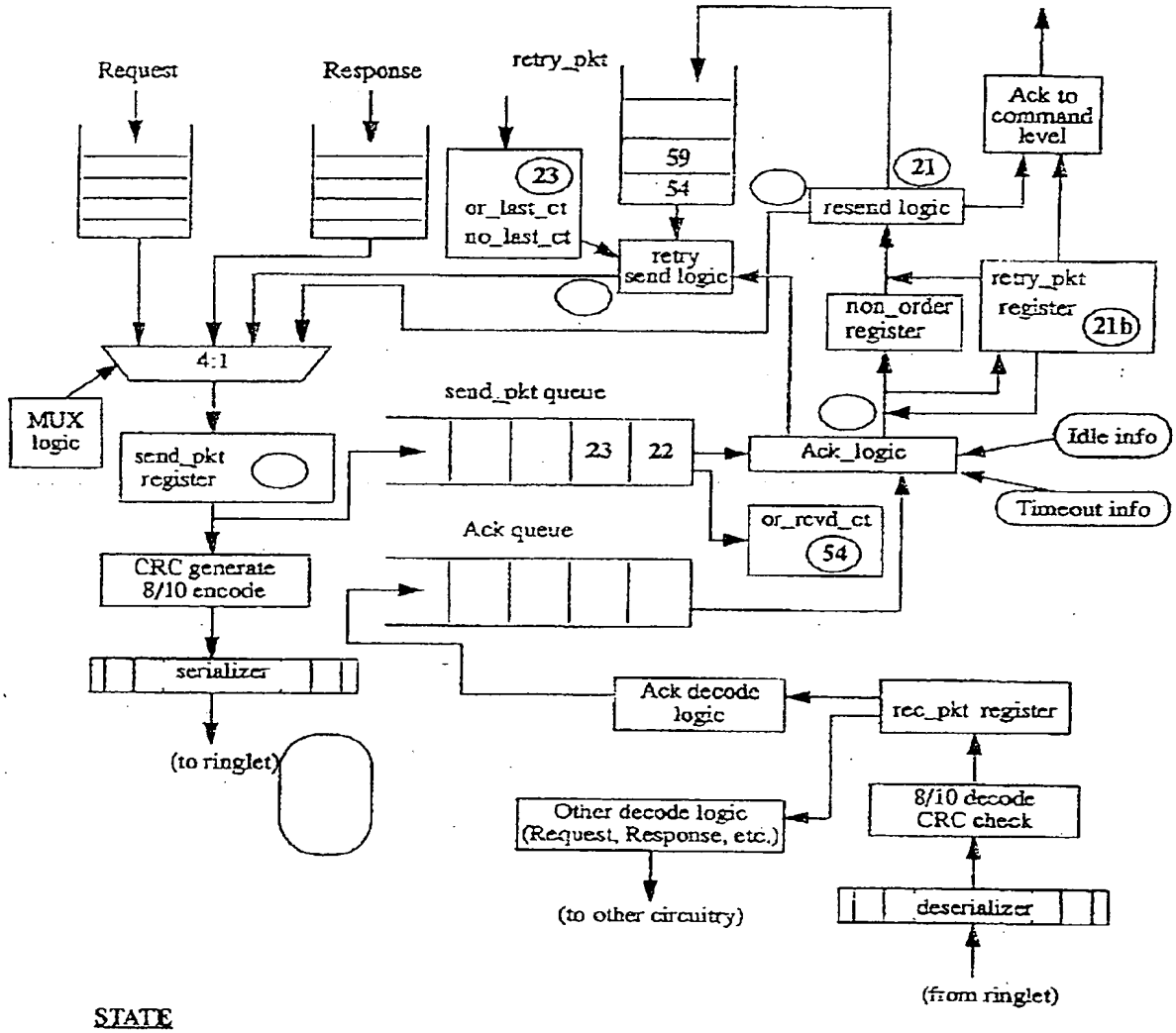
Fig 3



【Ｆｉｇ．３９】


Request
Response
retry_pkt
Ack to command level
or_last_ct
no_last_ct
retry send logic
resend logic
retry_pkt register
non_order register
4:1
MUX logic
send_pkt queue
send_pkt register
Ack_logic
Idle info
Timeout info
or_rcvd_ct
Ack queue
CRC generate 8/10 encode
serializer
(to ringlet)
Ack decode logic
rec_pkt register
Other decode logic (Request, Response, etc.)
8/10 decode CRC check
(to other circuitry)
deserializer
(from ringlet)

【Ｆｉｇ．４０】



STATE

•or_last_ct: last ordered packet sent

•or_rcvd_ct: ordered packet received count, frozen if busy retry required (detected in resend logic)

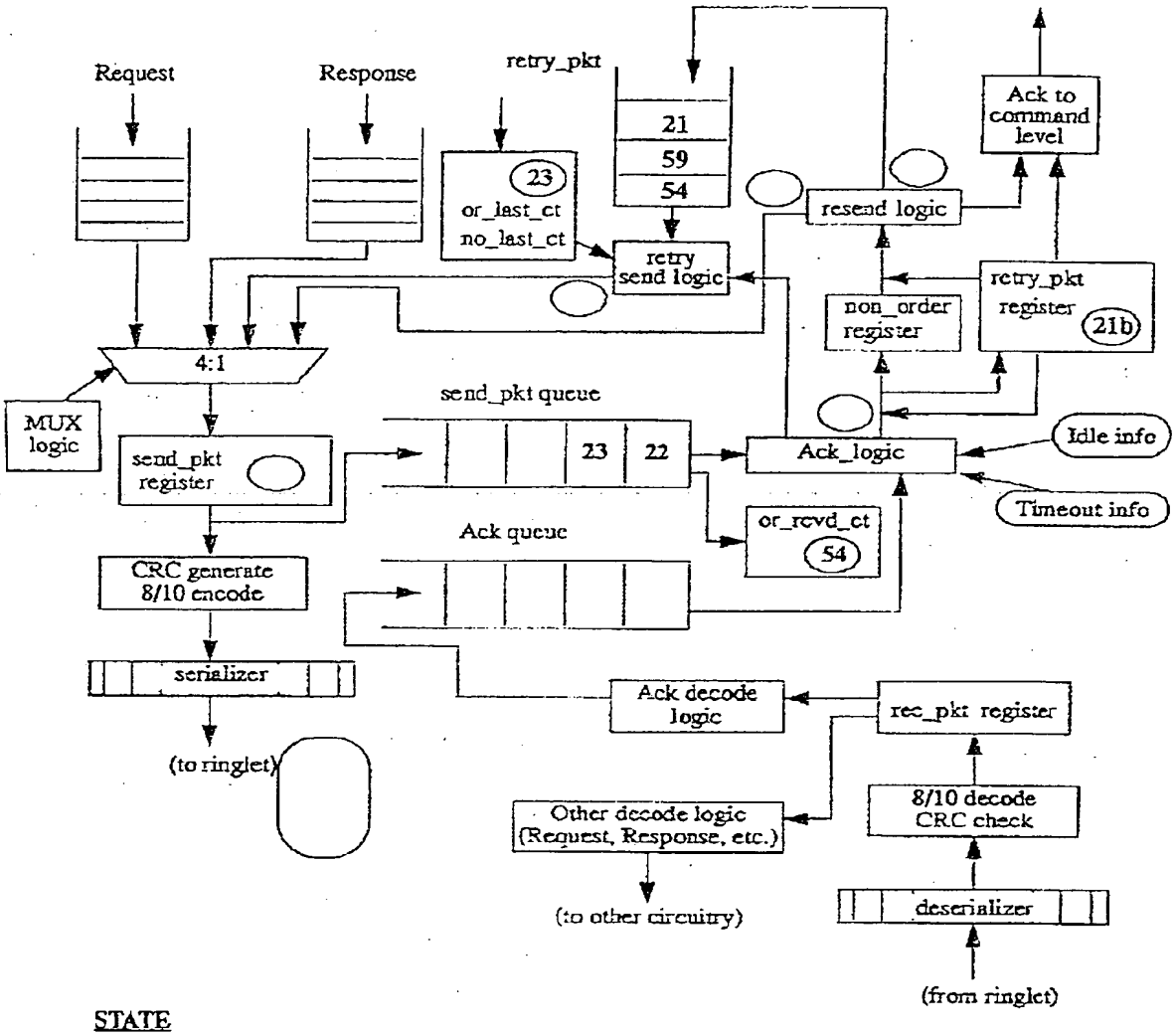
•Previous busy packets 54 and 59 (modulo 64) are in retry_packet queue.

Note: (or_last_ct - or_recd_ct) = (23-59) modulo 64
= 28, and 28 ≤ threshold for (or_last_ct - or_recd_ct) difference.

•Detect ack_busy in resend logic

•Add packet to retry_pkt queue

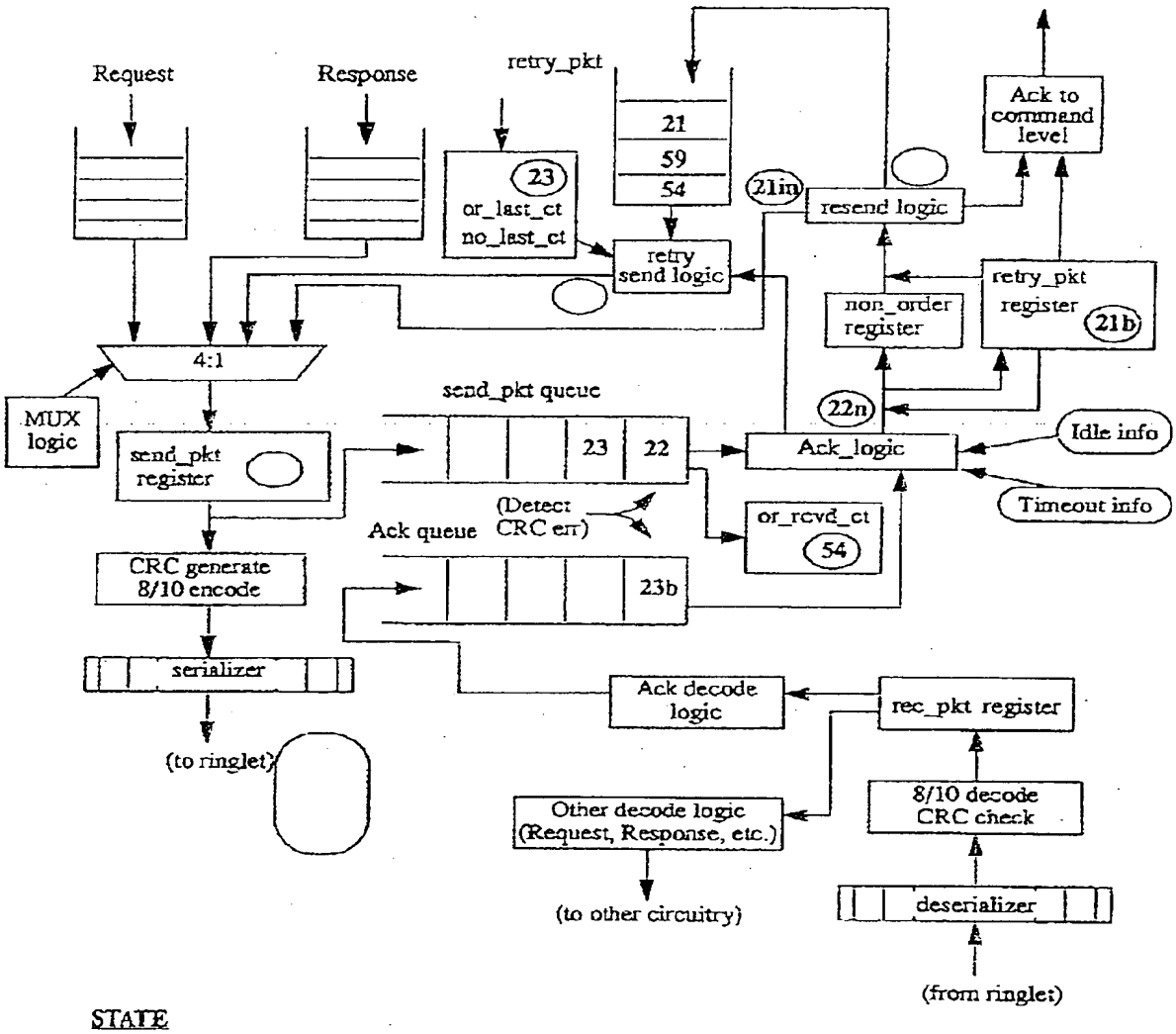
〔Ｆｉｇ．４１〕



STATE

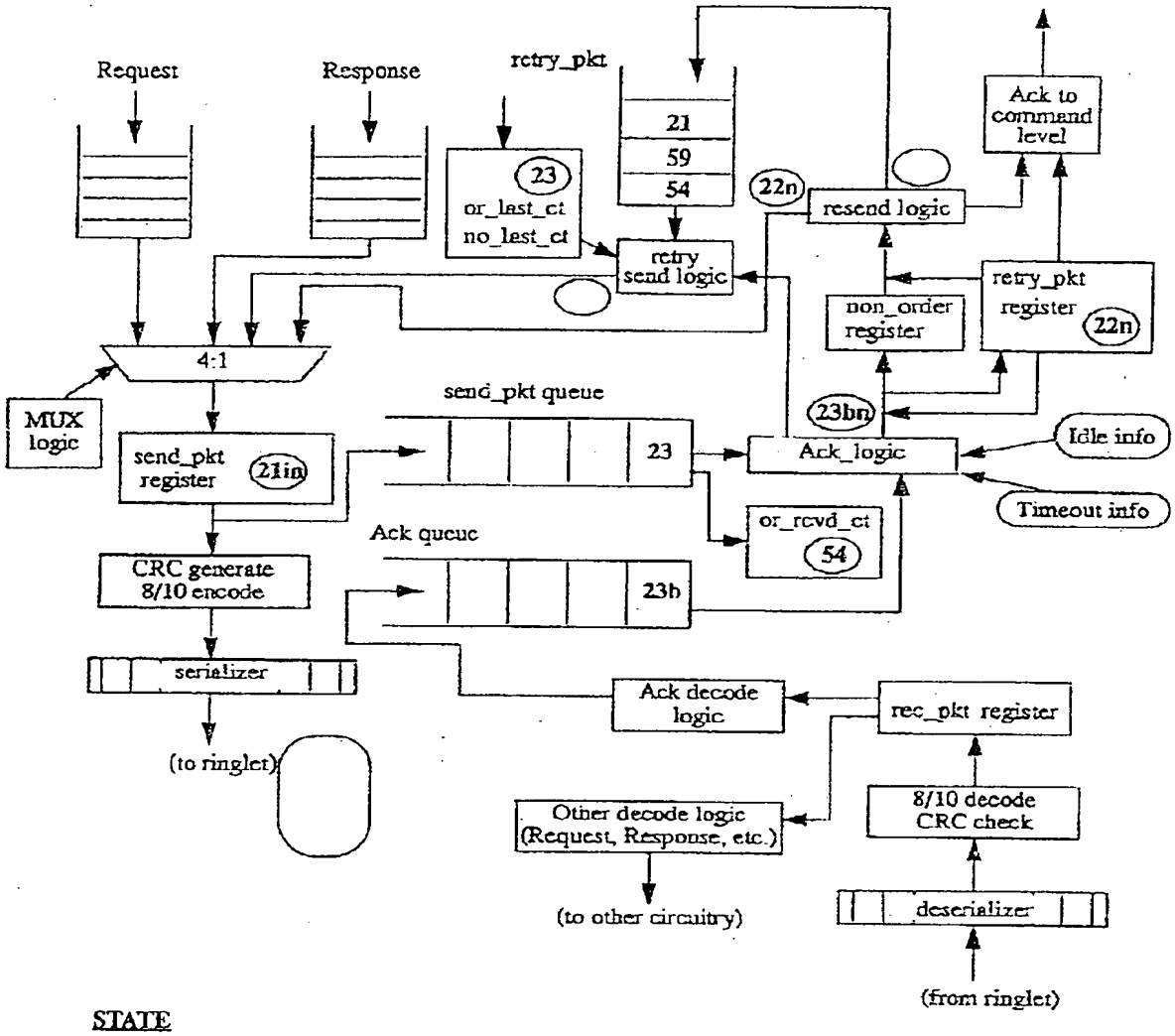
•Detect retry_pkt queue exceeds threshold. Set busy_loop.set$ and CRC_ack_chk

〔Ｆｉｇ．４２〕



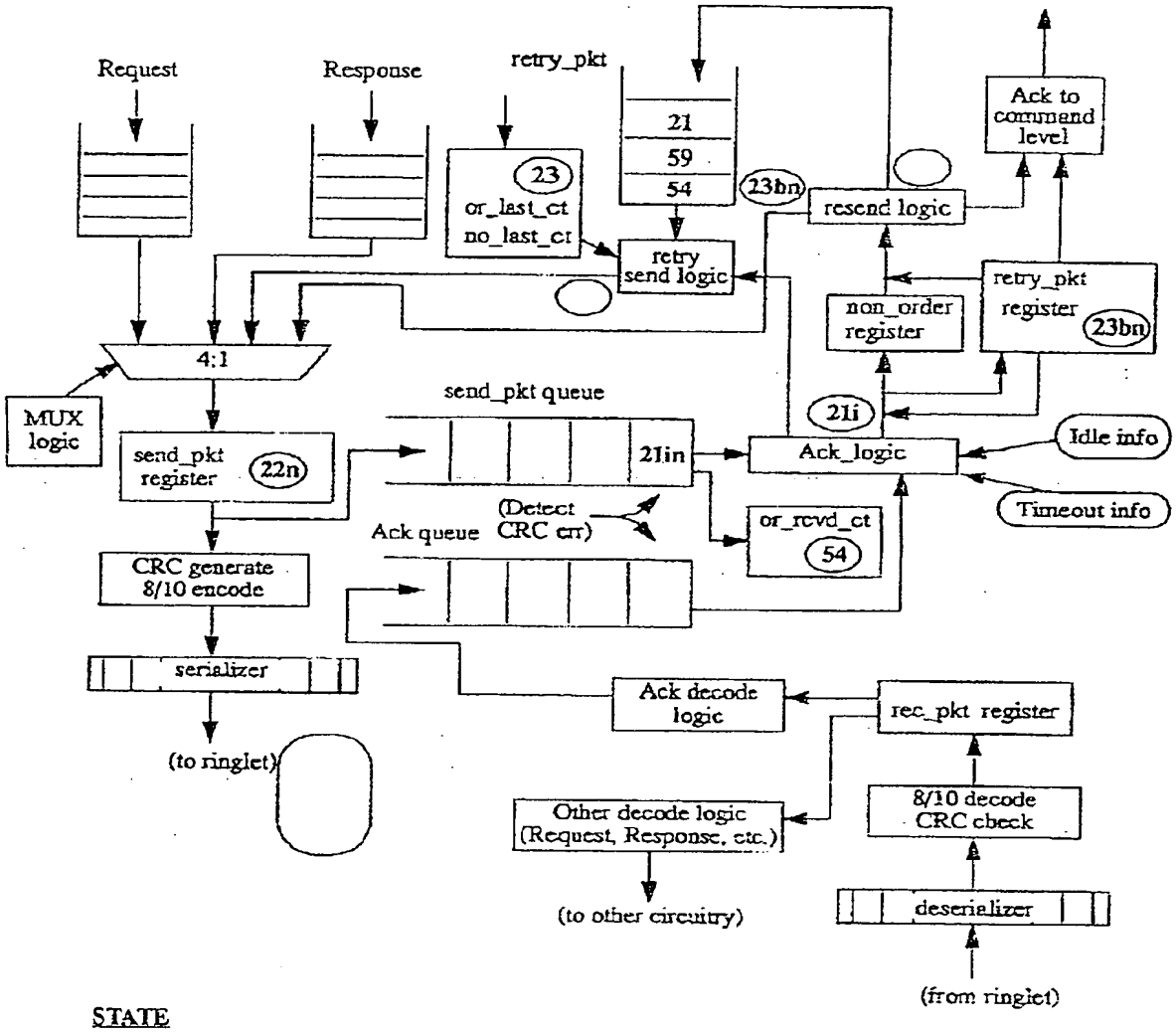
STATE
*Detect CRC error at head of send_pkt queue. Set CRC_err_retry control bit.
*CRC_ack_chk active.

【Ｆｉｇ．４３】



STATE
*CRC_ack_chk active.
*Packet 21 inhibited from retransmission on the ringlet.
*CRC_err_retry control bit active.

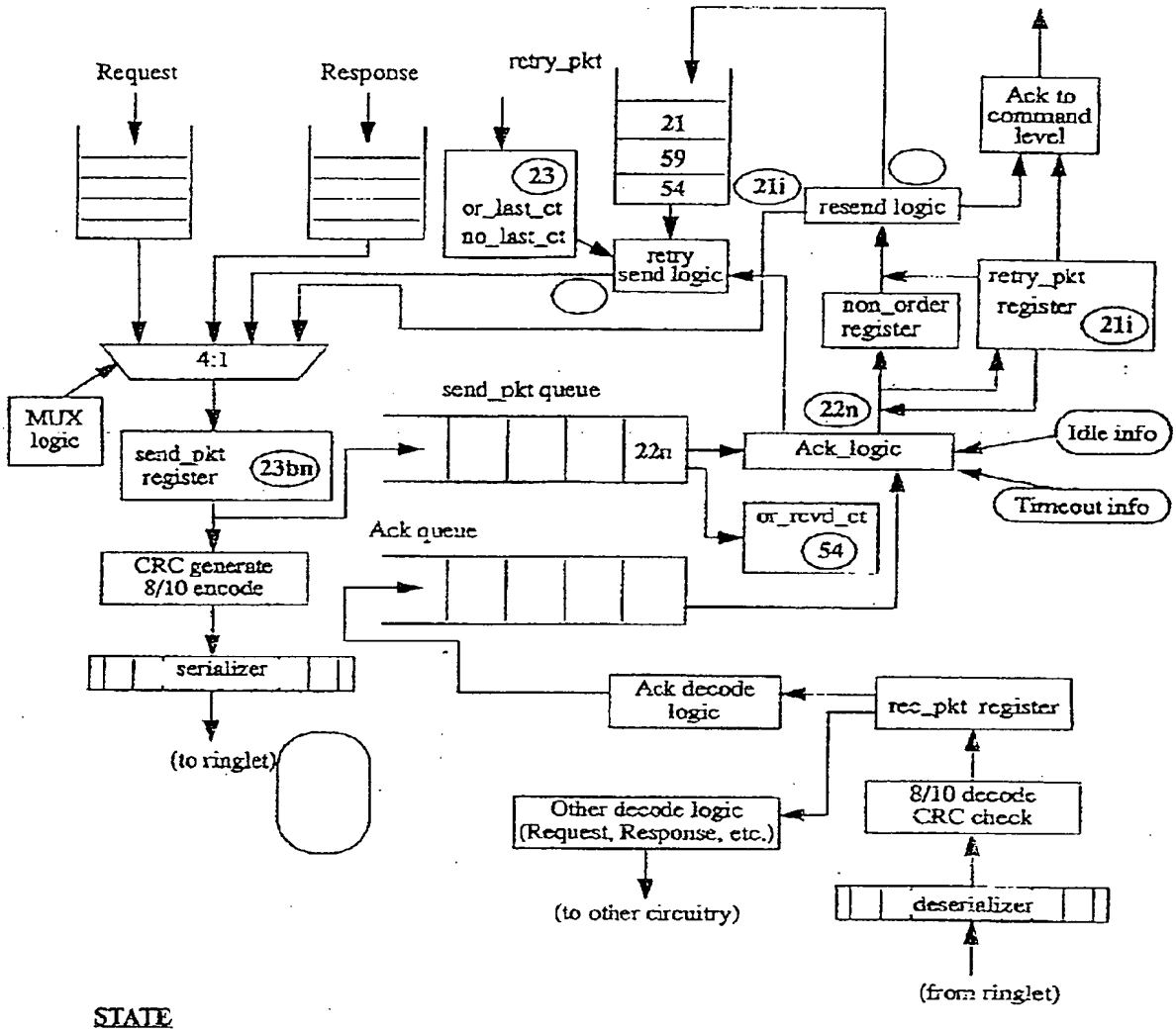
【Ｆｉｇ．４４】



STATE

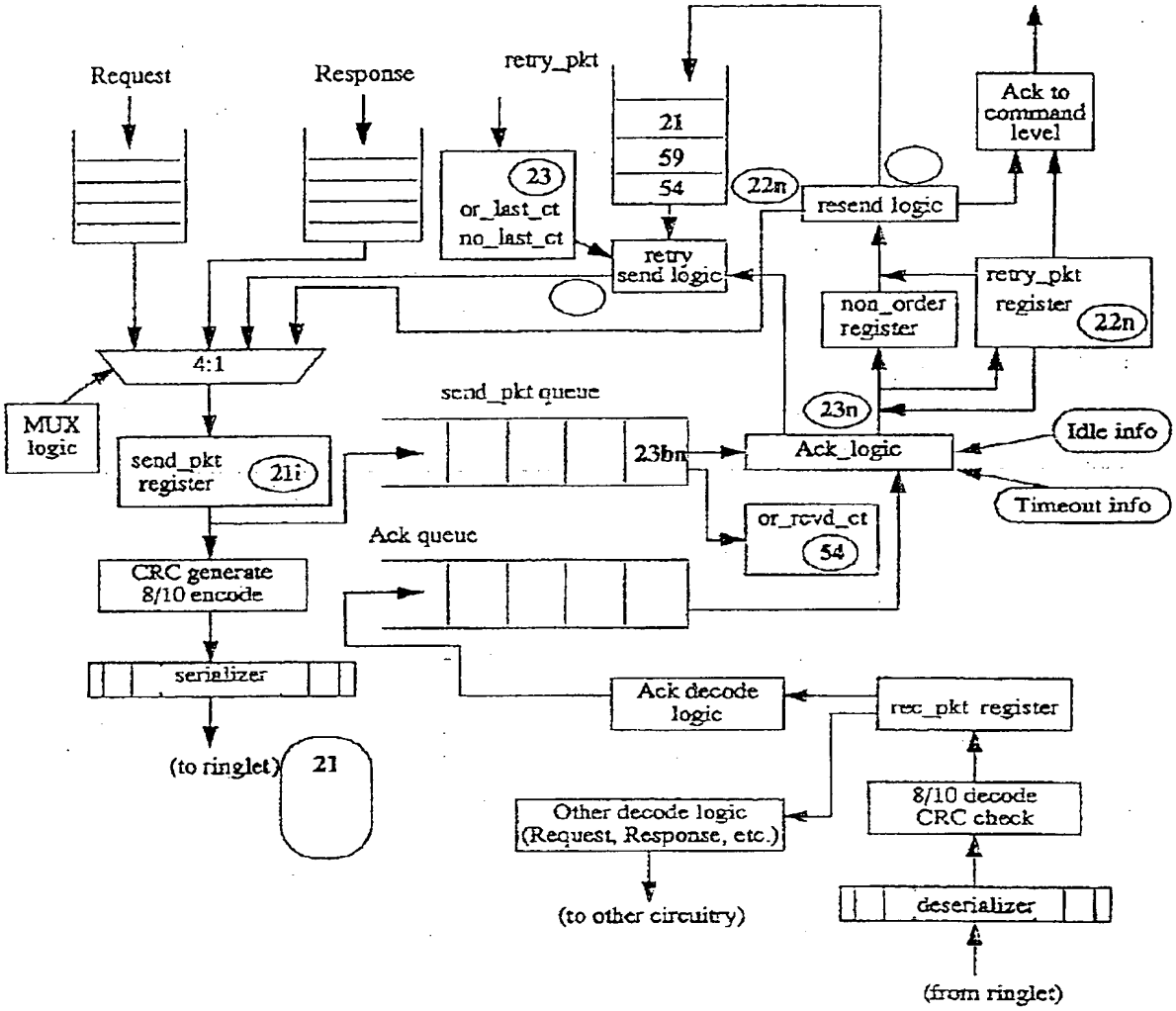
*CRC-ack-chk reset condition detected through "init_err" state at head of send_pkt queue
*CRC_init_err to be set because CRC_err_retry is active
*CRC_err_retry to be reset

【Ｆｉｇ．４５】



STATE
*CRC_init_err active

【Ｆｉｇ．４６】



STATE

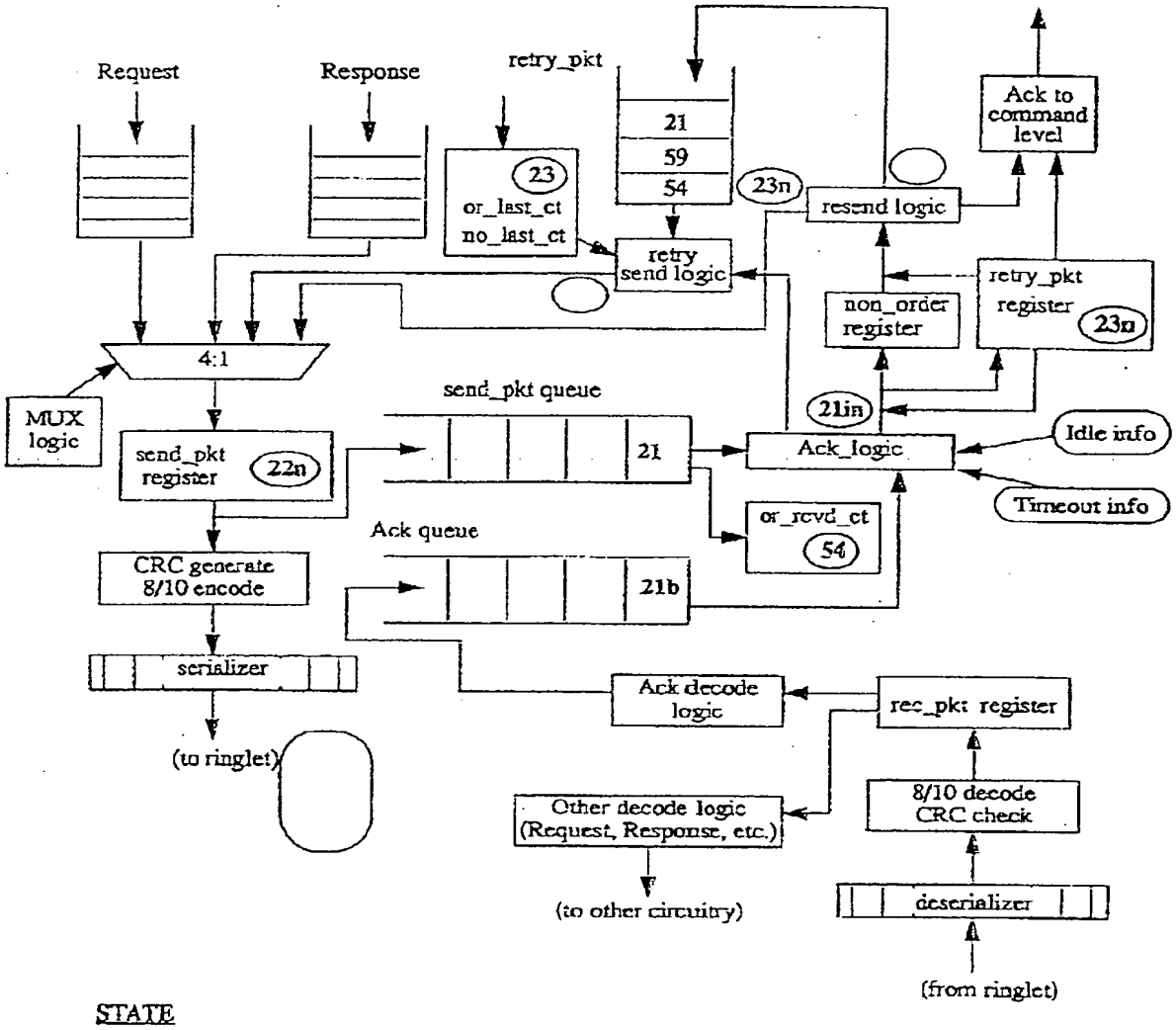
*Busy retry loop – pending

*CRC_init_err active

–There is no need to maintain "busy" acknowledgment in this state, since acknowledgment on the error retry packet will be used to determin "done" or "busy" packet disposition.

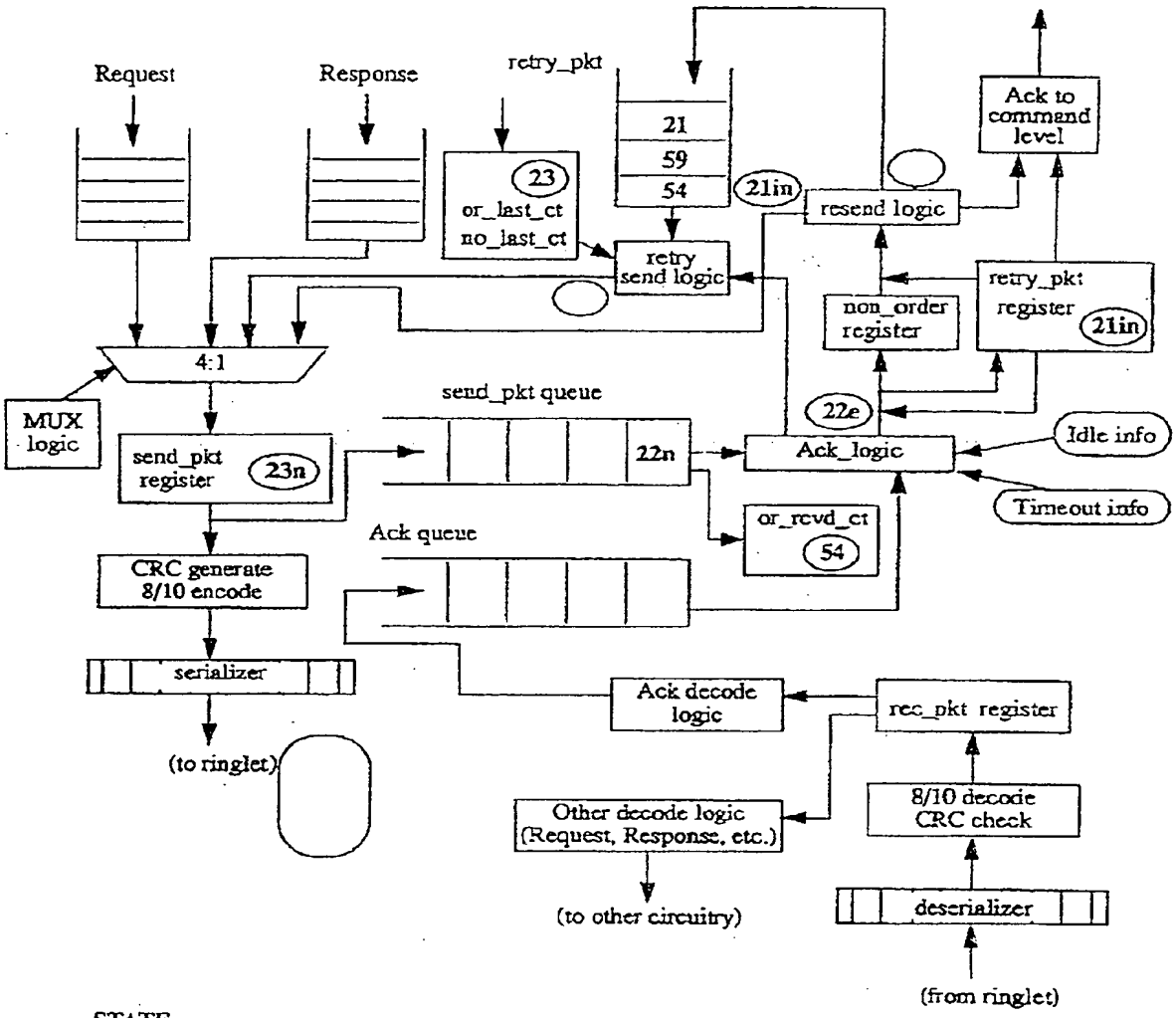
–"Known good packet" 21 is transmitted to the ringlet and enqueued in send_pkt queue to test acknowledgment.

[Fig. 47]



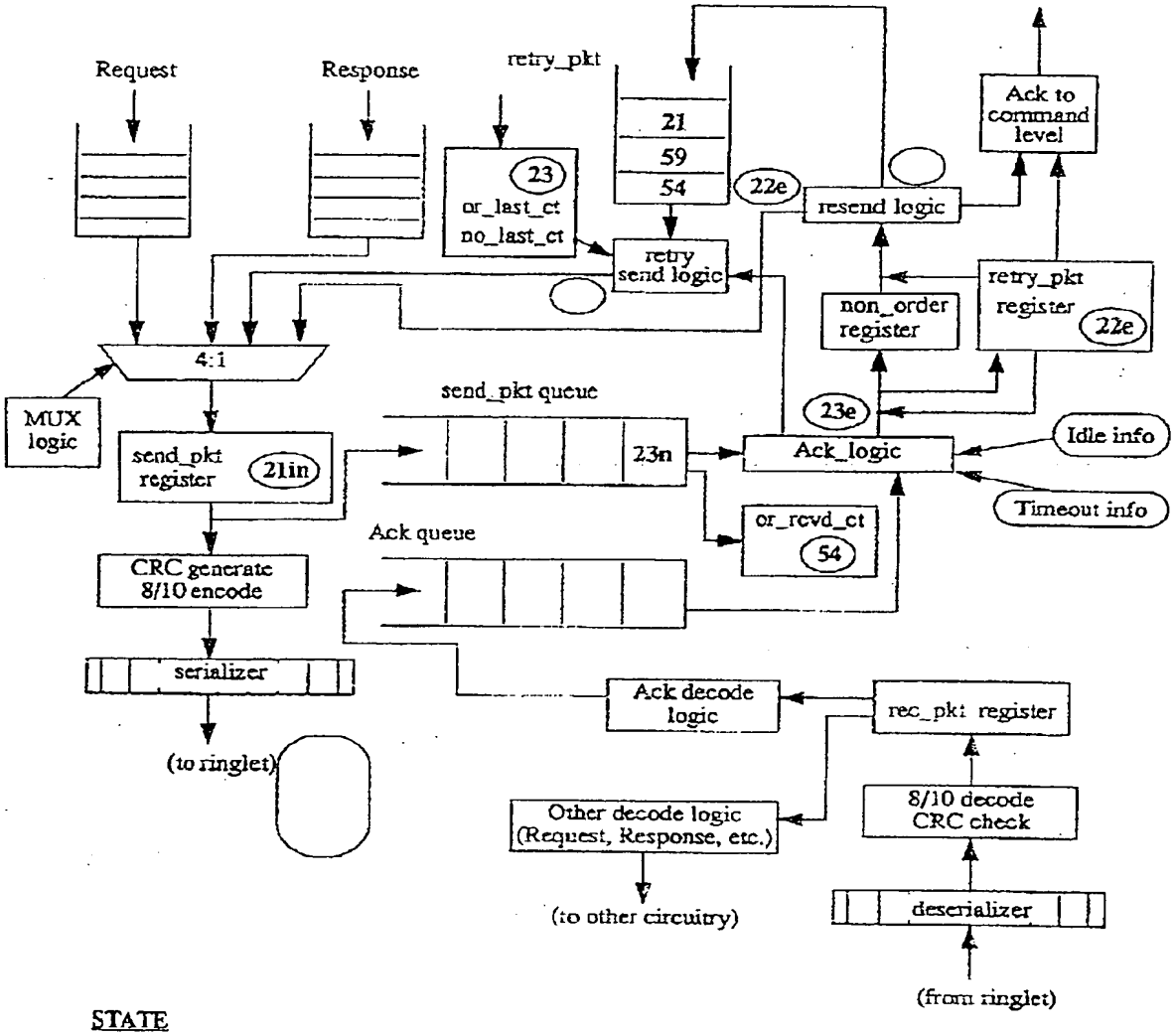
STATE
*Busy retry loop -- pending
*CRC_init_err is reset by detecting valid ack for "known good" packet 21.
*CRC_err is set.

【Ｆｉｇ．４８】



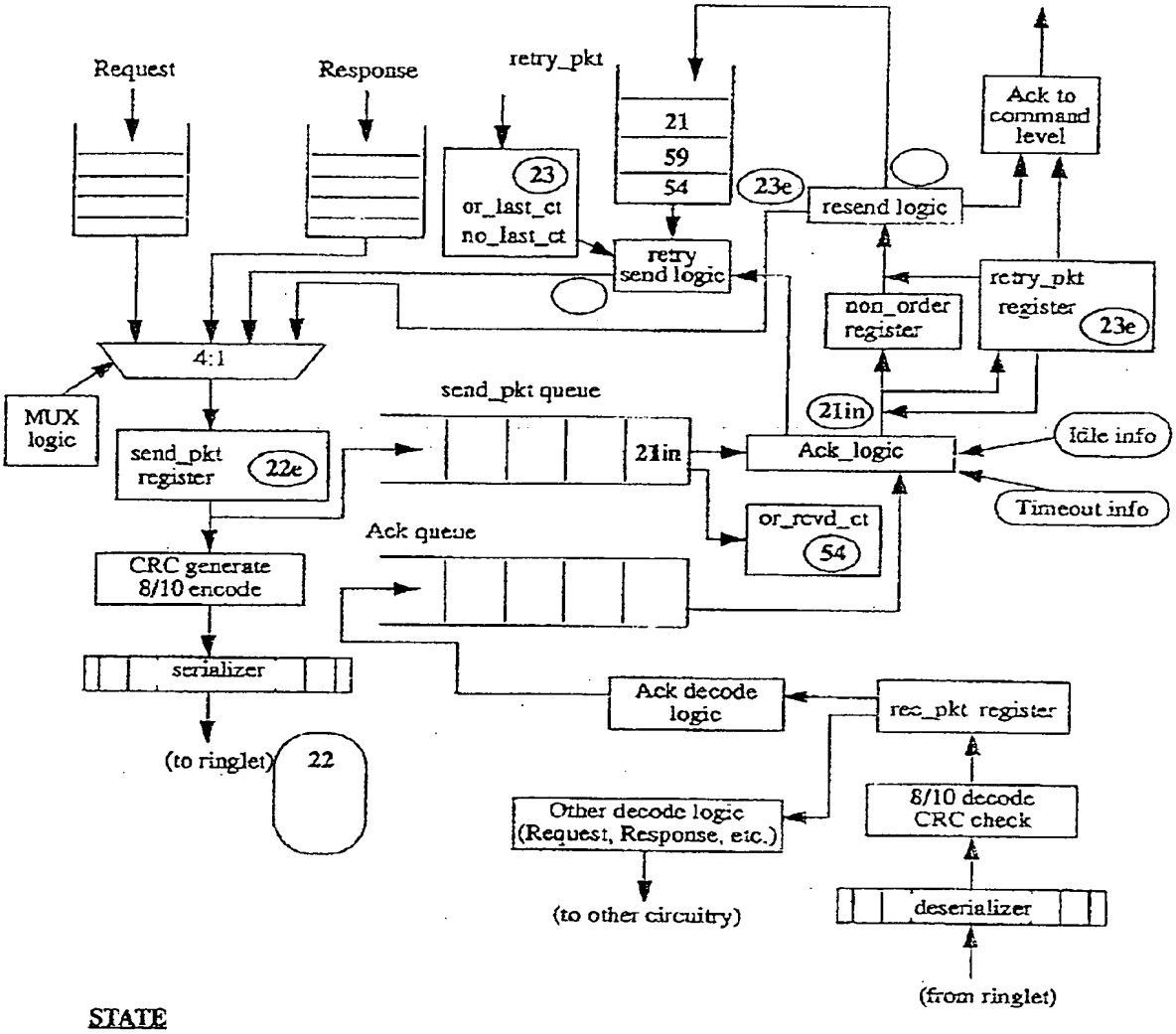
STATE
*Busy retry loop -- pending
*CRC_err active

[Fig. 49]



STATE
*Busy retry loop – pending
*CRC_err active
--Packet 21 is cycled through but is not resent (no xmit state)

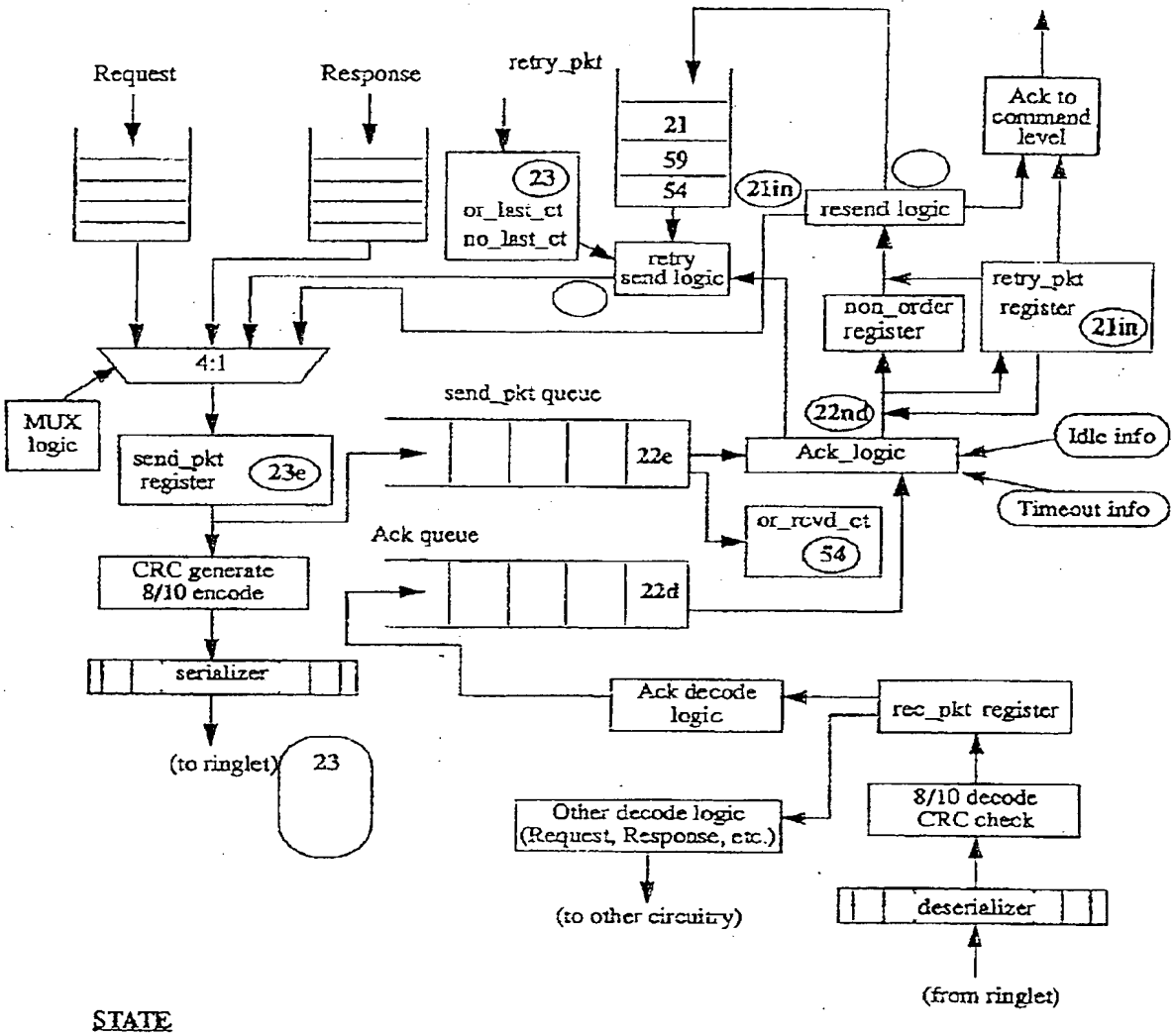
【Ｆｉｇ．５０】



STATE

*Busy retry loop – pending
*CRC_err is reset by detecting (in Ack_logic) "known good" packet 21 with ("init_err" and "no_xmit") state.
*CRC_ack_chk is started
*Packet 22 is retransmitted

[Fig. 51]



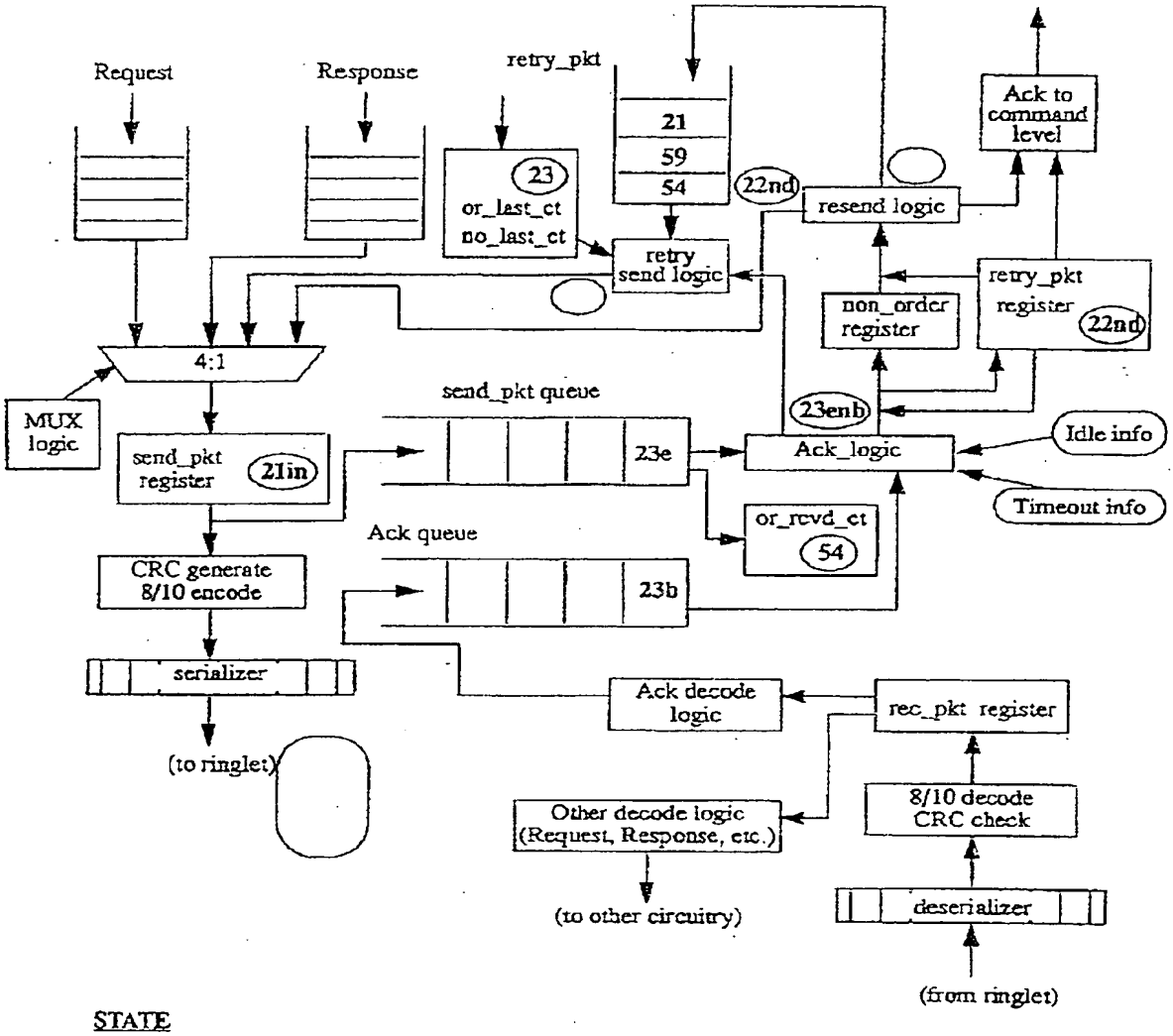
STATE

*Busy retry loop -- pending

*CRC_ack_chk is active

–Retried packet 22 gets "ack_done". Remaining "acks" in retry loop must be checked before 22 can be retired. "ack_done" state is saved.

【Ｆｉｇ．５２】



STATE

*Busy retry loop -- pending

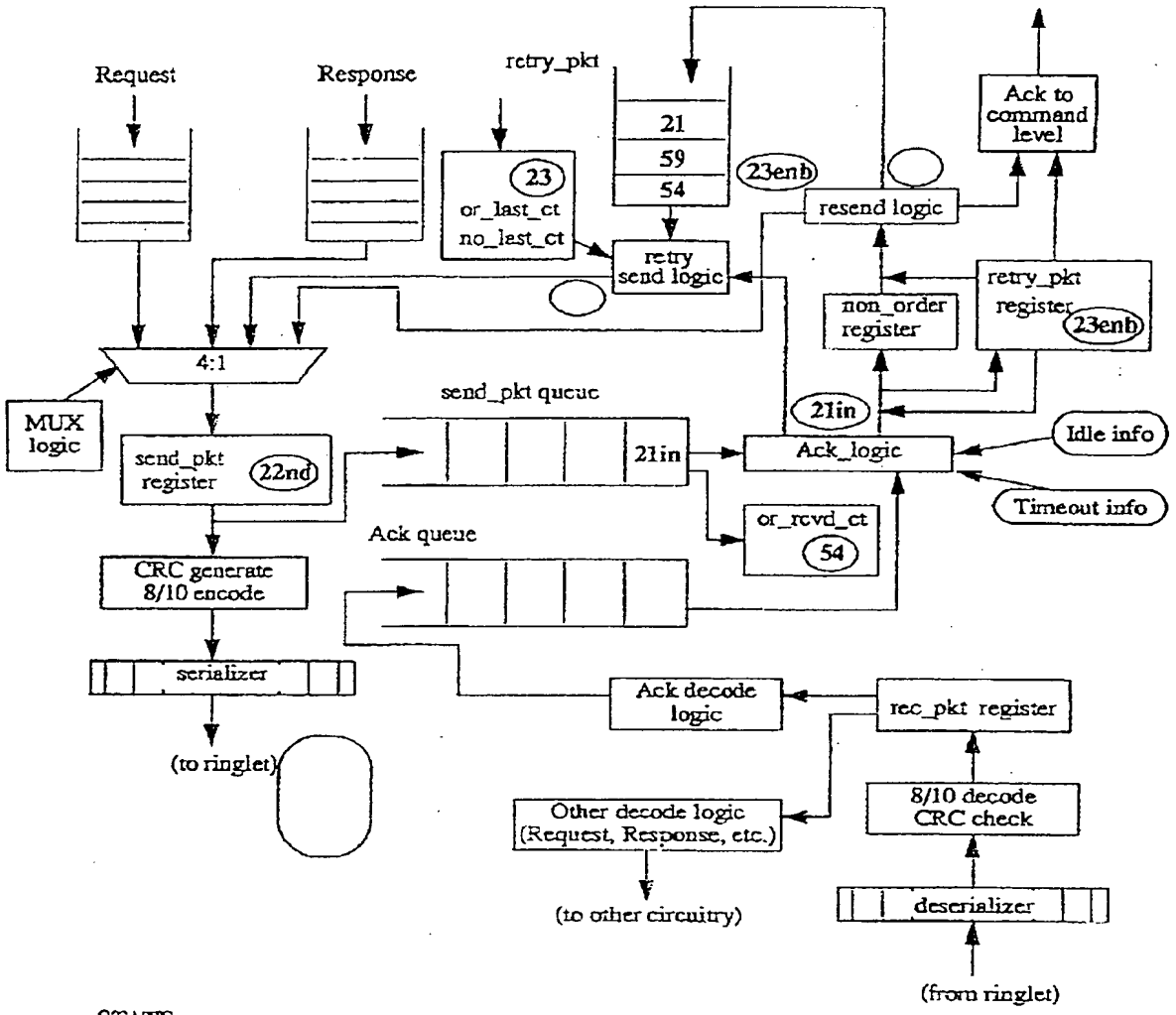
*CRC_ack_chk is active

--Retried packet 23 gets "ack_busy" again.

--Packet 21 is not retransmitted.

--Packet 23, the head of the send_pkt queue, is recognized as the last packet by comparing its seq field with the *(cr_last_ct) register. Its state is set to (crr_rct and no_xmit). Its busy acknowledgment is maintained with the "busy" state.

【Ｆｉｇ．５３】



STATE

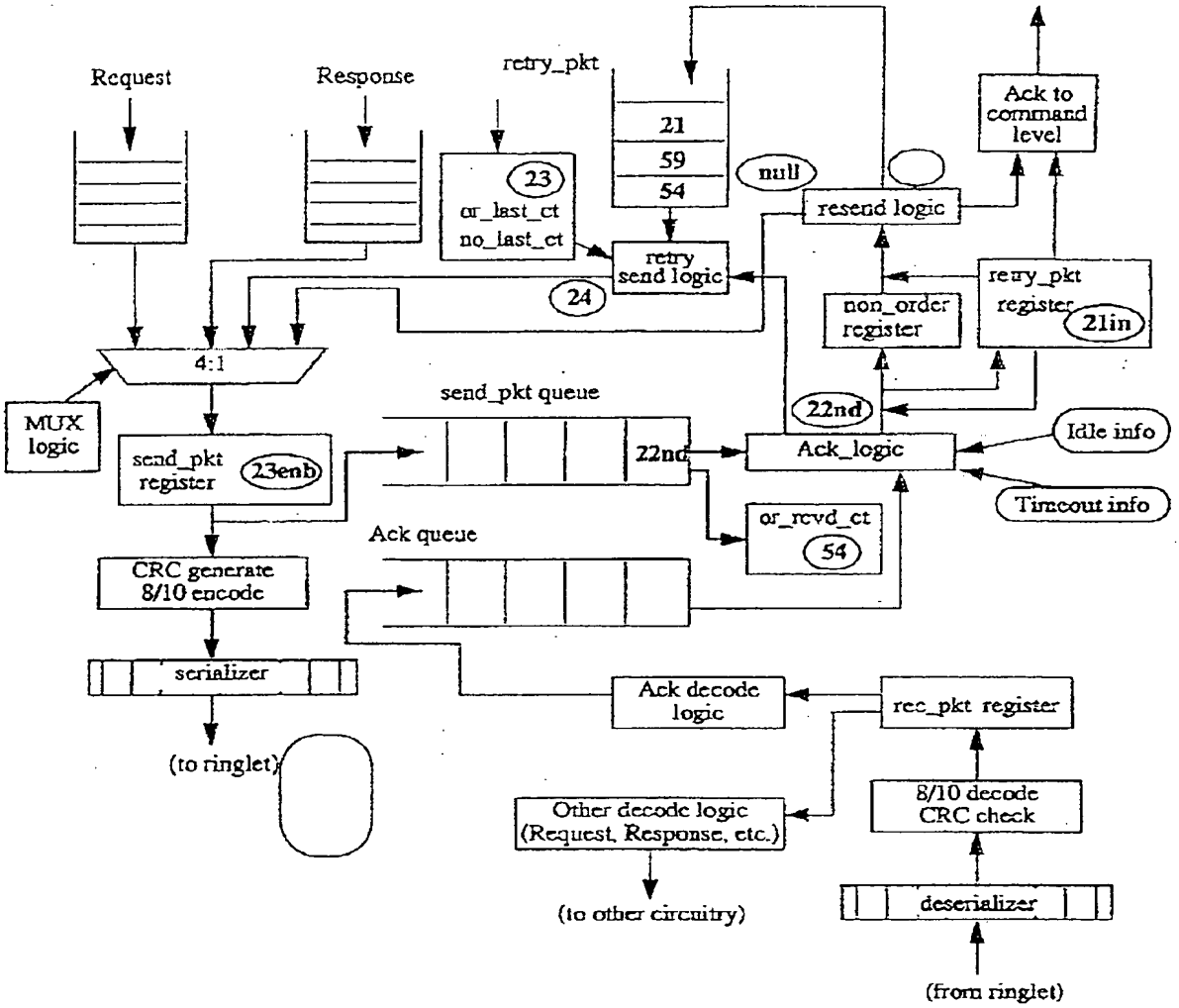
*Busy retry loop -- pending

*Reset CRC_ack_chk and busy_loop_pnd by detecting "known good" packet 21 with (init_err and no_xmit) state and no error detected during CRC_ack_chk loop.

--Retried packet 23 gets "ack_busy" again.

--Set CRC_err_end, residing in resend logic, to detect last packet when it cycles through retry_pkt_reg.

【Ｆｉｇ．５４】



STATE

*Busy retry loop starts

--Retry_send logic modifies seq field for head of retry_pkt to be (or_last_ct ÷ 1)

–Increment (or_last_ct) by 1.

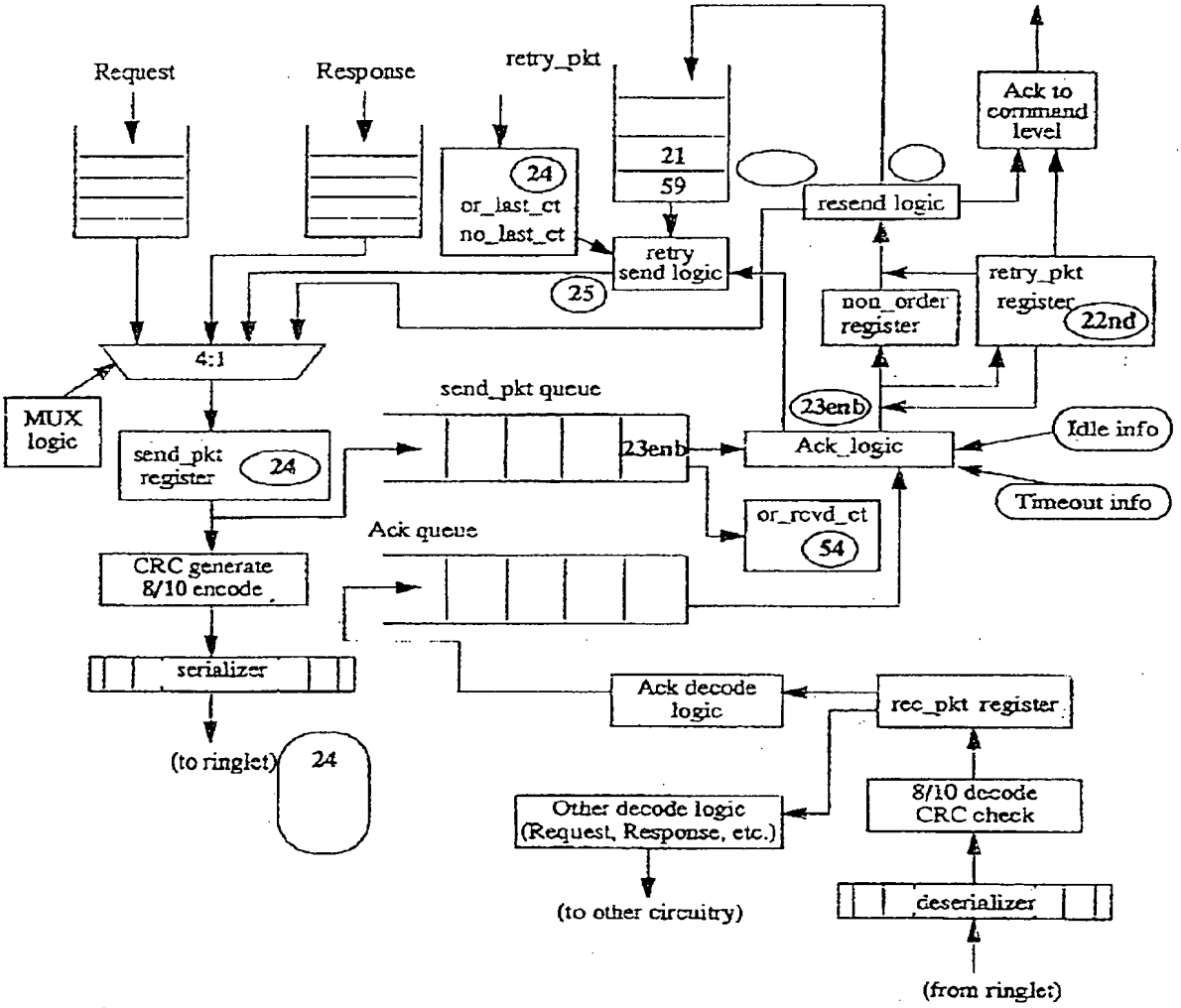
–Dequeue packet 54 from retry_pkt queue.

*CRC_err_end active in resend logic

--Drop "known good" packet

*Packet 24 assigned new "bz seq" field as first retried busy product

【Ｆｉｇ．５５】



STATE

*Busy retry loop active

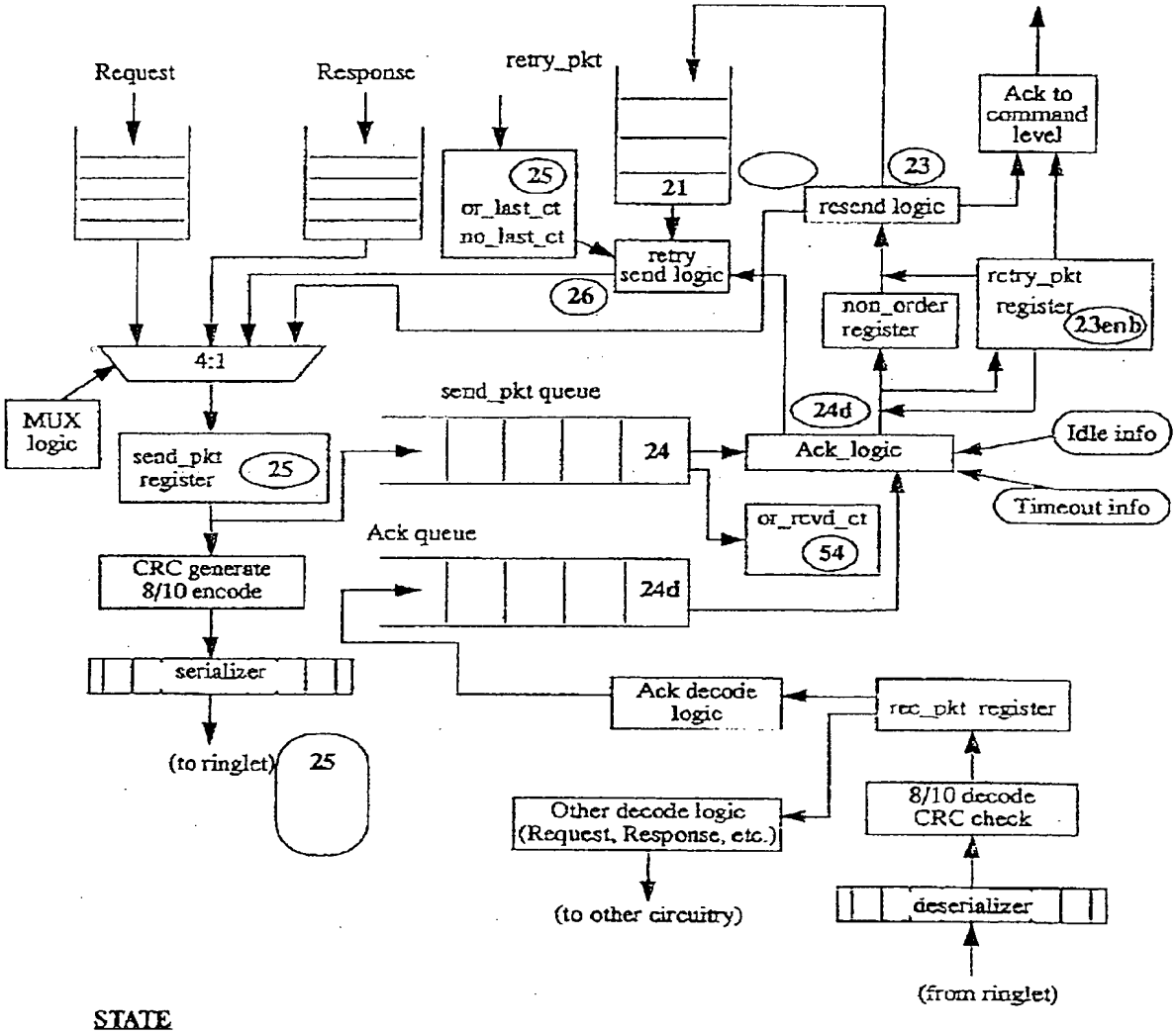
*CRC_err_end active in resend logic

--Packet 22 completed

--Packet 24 transmitted to ringlet

*Busy retry packet 59 assigned sequence number 25

【Ｆｉｇ．５６】



STATE

*Busy retry loop active

–Packet 24 "done" acknowledge detected

*CRC_err_end loop

–Packet 23 detected as last packet with "last packet" state (err_rst and no_xmit) and (busy or done).

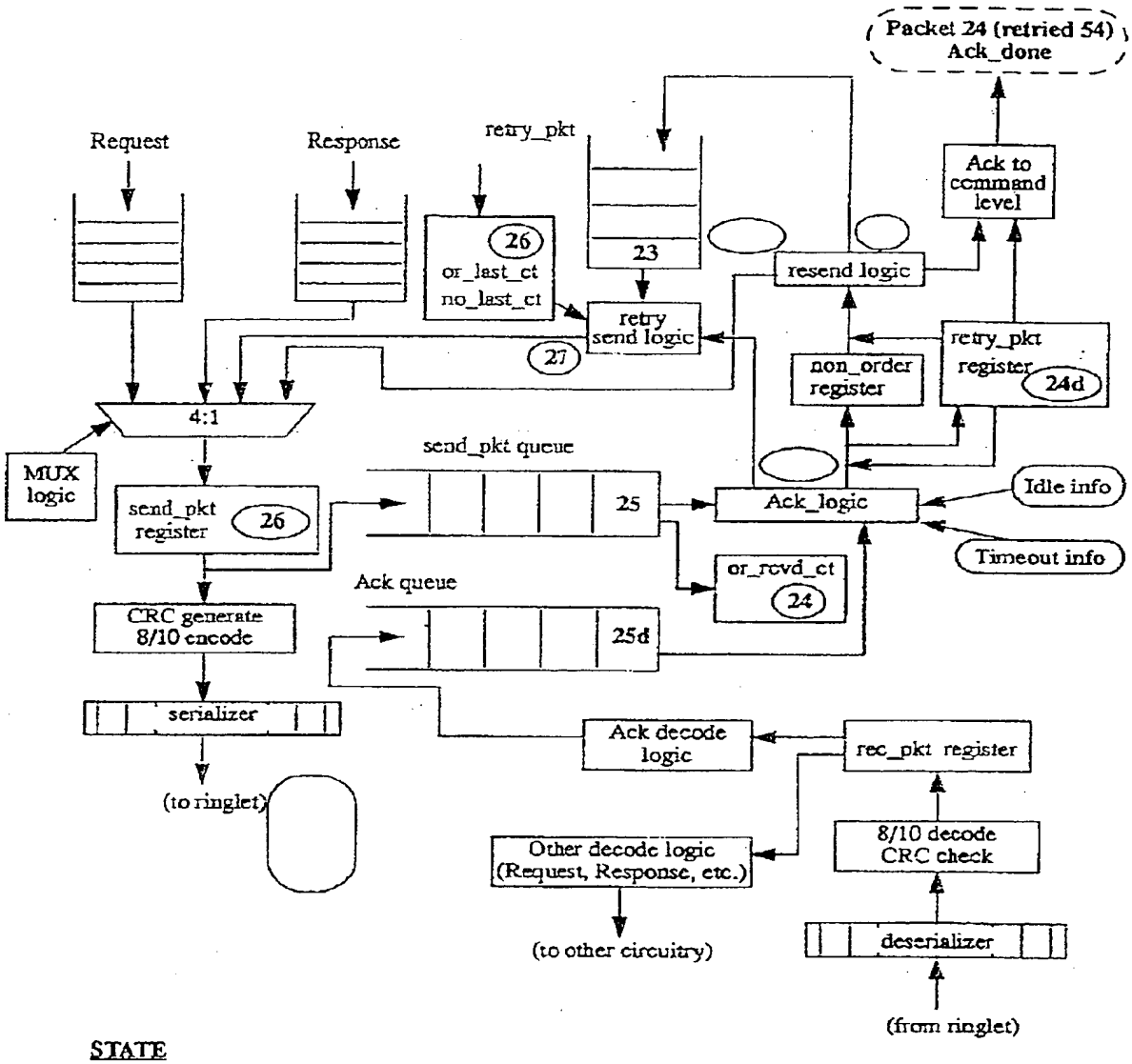
–"Last packet" state resets CRC_err_end loop

–"Last packet" state sets "retry_pkt_cmplt" control bit for busy retry loop. When "retry_pkt_cmplt" is set and the retry_pkt queue is empty, busy retry loop is reset.

*Retried packet 25 sent to ringlet.

*Busy retry packet 21 assigned sequence number 26

【Ｆｉｇ．５７】



STATE

*Busy retry loop active with "retry_pkt_cmplt"

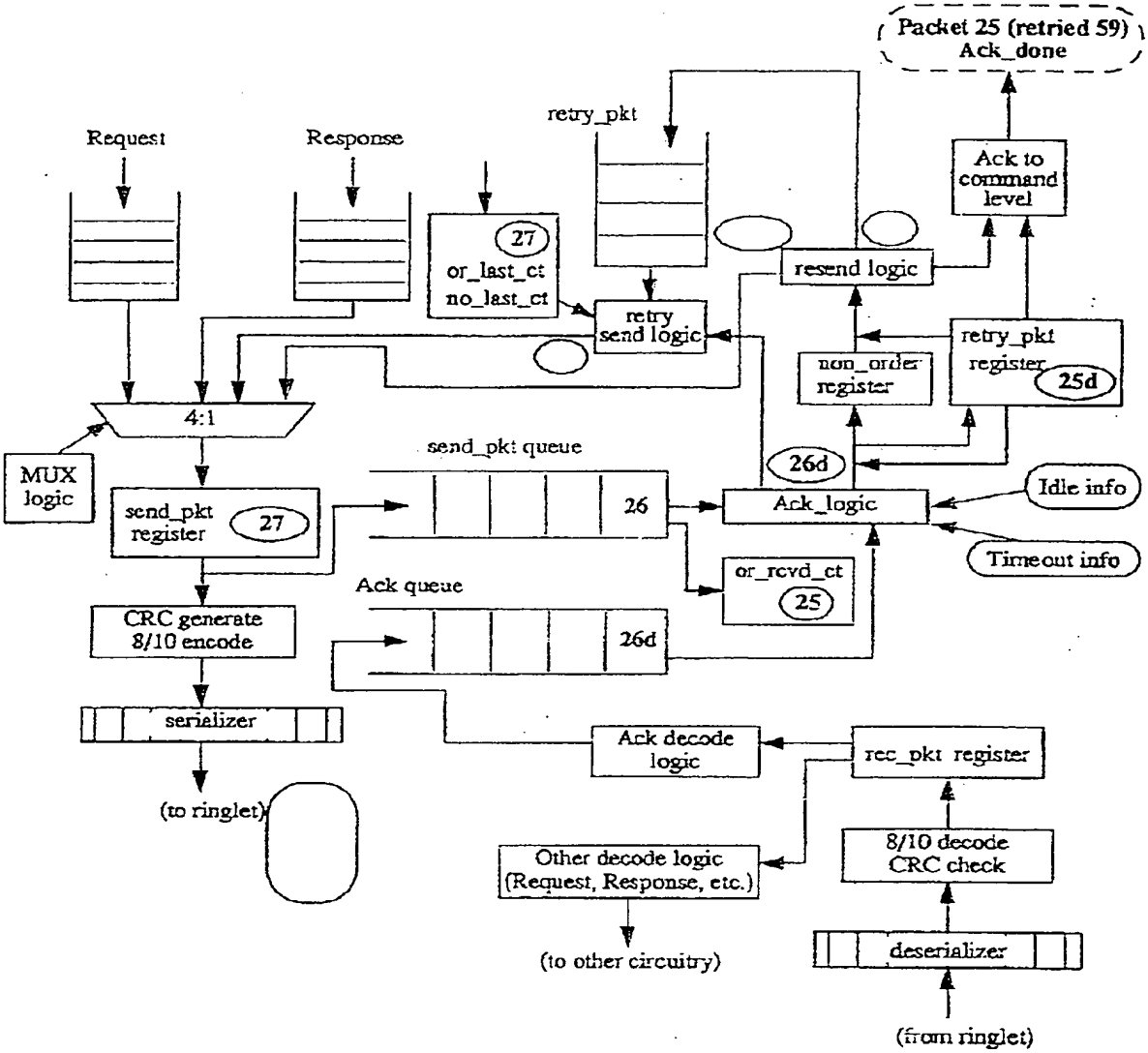
--Packet 23 in head of retry_pkt queue assigned new seq field 25 from (or_last_ct) + 1

--(or_last_ct) incremented by 1

--Packet 23 dequeued

*Packet 24 retried as ack_done

【Ｆｉｇ．５８】

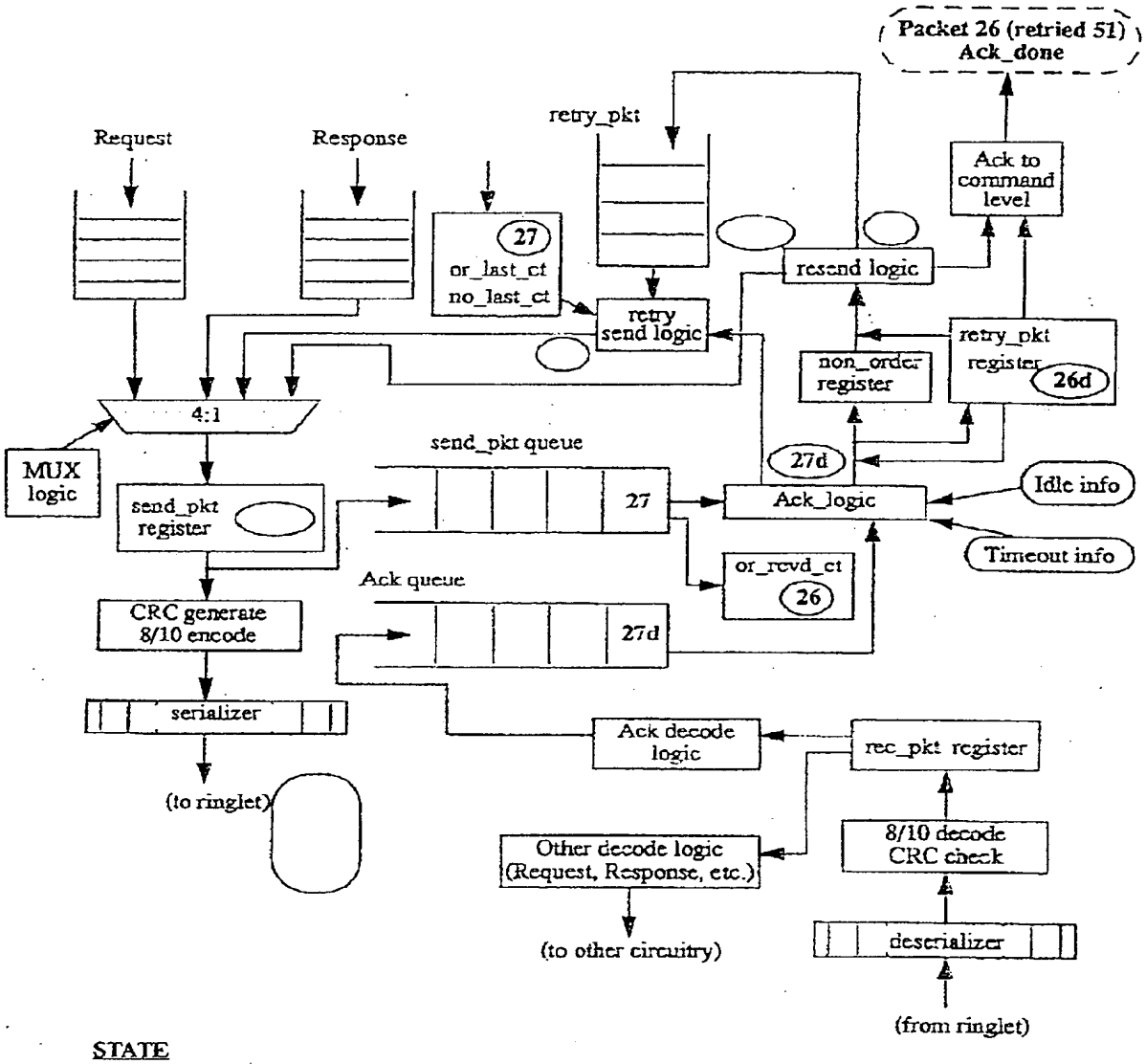


STATE

*Busy retry loop active with "retry_pkt_cmplt"

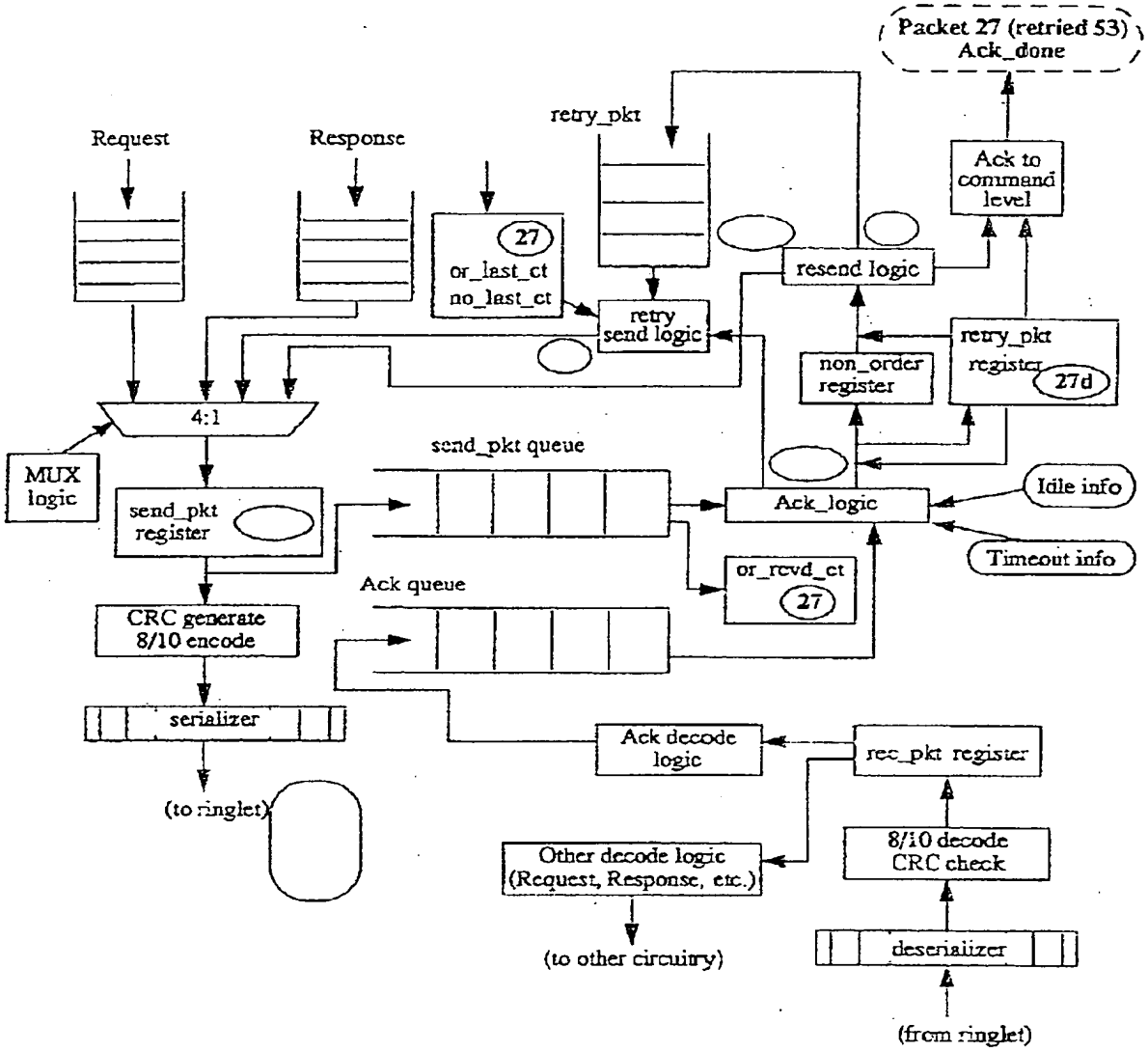
–Empty retry_pkt queue detected with "retry_pkt_cmplt" - resets busy retry loop and "retry_pkt_cmplt"

【Ｆｉｇ．５９】



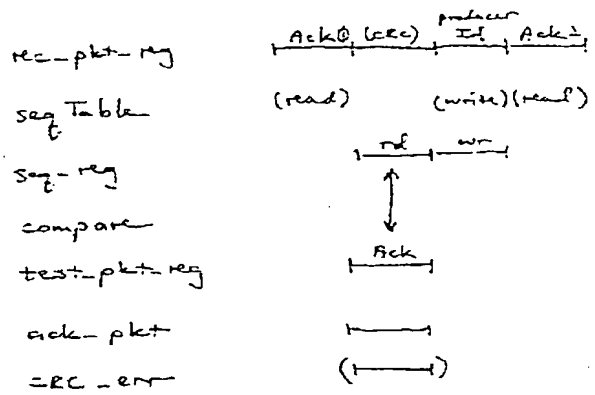
STATE
*None

〔Ｆｉｇ．６０〕



STATE
*None

【Ｆｉｇ．２７Ｂ】


Acknowledge timing (addressed to alternate node)
rec_pkt_reg
Ack0 (rec)
Ack1
seq Table
(read)
(write)(read)
seq_reg
rd
wr
compare
test_pkt_reg
Ack
ack_pkt
CRC_err

Abstract of the Disclosure

A system for maintaining reliable packet distribution in a ring network with support for strongly ordered, nonidempotent commands. Each consumer node on the network maintains a record of the sequence of packets that have passed through that node, and the state of each of the packets at the time it passed through, including a record of the last known good packet and its sequence number. When a producer node detects an error condition in an acknowledgment for a packet, resends all packets beginning with the last known good packet. Each consumer node is able to process or reject the resent packets, including packets that may already have been processed, which it is aware of due to the packet and state records for all packets. Strong ordering is thus supported, since the sending order of packets can be maintained for processing due to the resends, and nonidempotent commands are supported due to the consumer nodes' ability to determine whether they have already processed a given packet, and to send an acknowledge-done reply if that is the case. The system is equipped to operate successfully in the presence of a failed or very busy node by maintaining a queue of busy acks from each node, or from nodes that are particularly busy, and when the queue is full taking steps to bypass the busy or failed node. Alternatively, the system can detect that some predetermined period of time has passed during which either no response or only busy responses have been received from a given node, and at that point determine to effectively bypass the node for future transactions, at least for some timeout period, so that packet transactions for the rest of the network can proceed unhindered by the ill-behaved node.